3 Programming Concurrent Systems

Butler Lampson

Abstract

Concurrent programs are nondeterministic, which makes them difficult to reason about and to debug. But today computing systems have lots of processors, so concurrent programs are unavoidable. Atomicity makes it possible to reason as though most of the program is sequential, with locks and conditions allowing a number of sequential processes to communicate safely using ideas introduced by Dijkstra in the 1960s.

3.1 Introduction

I first met Edsger Dijkstra at the first ACM Symposium on Operating System Principles in Gatlinburg, TN, in early October of 1967. This was one week after I got married. He presented a paper on the THE system [Dijkstra 1968b], which became an instant classic. Together with earlier papers that showed how to implement mutual exclusion using only atomic load and store instructions [Dijkstra 1965] and how to use semaphores to program reliably with multiple concurrent processes [Dijkstra 1968a], it introduced the ideas of sequential processes, nondeterminism, mutual exclusion, and producer–consumer synchronization that are still the basis for concurrent programming today. These ideas developed into the notion of atomic actions as the way to reason sequentially as much as possible.

These papers were far in advance of their time. Dijkstra wrote that he was motivated by fear of the nondeterminism of concurrency and wanted to prove that it was tamed in his programs. It was a while before the ideas took hold because at the time machines had only one CPU and user processes normally communicated only through the file system, if at all, so that the only tricky concurrency came from I/O channels and their interrupt routines. Kernel calls and interrupt routines ran to completion, and mutual exclusion was done by disabling interrupts. In this simple setting, willpower was usually sufficient to tame the concurrency and proof seemed unnecessary to most people building systems, including myself.

When you have several processors, or a highly interactive system like a PC with windows, there is a lot more concurrency with shared variables, especially if there is just one address space so that it's easy for processes to share memory. Operating systems in the 1960s and early 1970s had none of these, but by the late 1970s Dijkstra's work began to seriously influence most system builders, often through Brinch Hansen's and Hoare's work on monitors [Hoare 1974, Lampson and Redell 1980].

When computer networks came along in the early 1970s, distributed systems became interesting. They have concurrency, but they also have partial failures and nonnegligible communication costs. As usual, Dijkstra was ahead of the game in stating and solving the problem of self-stabilization [Dijkstra 1974], but by that time he was no longer building systems himself and his main interests were elsewhere.

There's a lot of demand for concurrent programs, since they're the only way to take advantage of multicore processors and distributed systems. Hence there is a lot to say about designing them. This chapter can only touch on the highlights: nondeterminism and atomicity.

3.2 Concurrency

The way to reason about an arbitrary computer system is to view it as a state machine, with state transitions or *actions* that take the system from one state to the next. A program defines the actions; it consists of *commands* that denote actions, and syntax that defines the order in which commands run. In this context, a correctness proof is based on an *invariant*, a state predicate that is always true.

Sequential systems are what we understand: the state doesn't change between one command and the next in the program. Thus if c_2 is the next command after c_1, and c_1 leaves the system in state s, then c_2 starts with the system in state s. This means that you can reason from the static program text about the system's *behavior*, the infinite number of sequences of states that it can exhibit.

The most common formal statement of this property is Hoare logic. Suppose P and Q are state predicates and c is a command. $\{P\}\ c\ \{Q\}$ means that if state s satisfies P and c takes s to s', then s' satisfies Q. In a sequential program, if $\{P\}\ c_1\ \{R\}$ and $\{R\}\ c_2\ \{Q\}$ then R expresses what we know about the state after the first command, and it doesn't change between the two commands, so it follows that $\{P\}\ c_1; c_2\ \{Q\}$; likewise for other ways to write successive commands. By repeating this reasoning, you can find a precondition and postcondition for any sequential program.

In a concurrent system there is more than one sequential agent taking actions; an agent might be a CPU, input–output device, network, or human user. The abstraction of a sequential agent is called a *sequential process*. In general, processes are *asynchronous*: the relative speed of two different processes is unknown and variable. This means that the behavior of the whole system is highly nondeterministic since the actions of the different processes can be interleaved in many different ways. After one process acts, any process might act next, so that the state may change between one command in a program and the next. It's hard to reason about this nondeterminism. The only way to avoid it completely is to give up asynchrony and tightly synchronize the processes so that the program determines exactly which process will act next. But this defeats the purpose of concurrency since a slow process can delay a fast one if you don't know the relative speeds.

There are *synchronous* concurrent systems in which several processes run (more or less) in lockstep, often executing the same instruction on different data. These are used for numerical computing and its close relatives, learning and graphics. They are outside the scope of this chapter.

3.2.1 Language Constructs for Concurrency

There are many ways to write a concurrent program:

- An explicit **fork** *r* of a new process that runs routine *r* concurrently, returning a *promise p*; then **await** *p* returns the result of *r*. Languages package this in many different ways.
- A **parbegin** $b_1; \ldots; b_n$ **parend** that runs the *n* blocks concurrently.
- A library such as a database or graphics system whose code runs concurrently, even though its caller is sequential.
- A distributed system, where all the machines run in parallel.

Everything here applies to all of these, except for locks in a distributed system, which don't work well because they don't play nicely with partial failures. A workaround is locks that time out, called *leases*, but distributed systems tend to avoid locks and rely on sharding and streaming.

3.3 Why Concurrency?

Concurrency began in operating systems (and their close relatives, real-time systems) because an operating system (OS) has to deal with I/O devices even when there is only one CPU. The only way to avoid concurrency is for the CPU to wait for each I/O operation to complete. This wastes a lot of CPU resources because a typical I/O takes many CPU cycles, but early machines like the IBM 704 did their I/O this way. And if there are application processes doing different things (and sometimes waiting for I/O, or perhaps interacting with different users), each one needs its own process or logical CPU, created by "time slicing" the single real CPU among the several logical ones. Such a logical CPU is also sometimes called a *virtual machine;* if it shares an address space and other resources with other logical CPUs it's often called a *thread* (and the address space is confusingly called a process).

Today, concurrency within a single application is also common because single-stream general-purpose processors are not getting much faster (since speeding up the clock makes the chip too hot [Leiserson et al. 2020], and there are only three ways to speed up a computation:

- Better algorithms or tighter code that takes fewer instructions or cache misses.
- Specialized hardware that runs inner loops faster.
- Concurrency that exploits multiple cores or multiple machines.

Only concurrency is reasonably general-purpose, and modern CPUs have multiple cores. With k cores, $n \geq k$ processes can do k times as much work as one process, but to use the cores reliably requires taming the nondeterminism. And to use them efficiently requires locality. For a computation to run fast, its data must be either immutable or local, because when a remote variable changes, getting its current value is costly—a cache miss on a multicore CPU takes as much time as several hundred instructions, and a network round trip is worse. Fast computations need P&L: parallelism and locality. With non-CPU agents, locality is usually not an issue, but nondeterminism definitely is.

A distributed system (multiple machines connected by a network) can have even more concurrency, at the expense of at least 100 times greater communication latency. It also has *partial failures*: some of the parts can fail, including the network links, while the whole system keeps working. You don't want to crash it all when one part fails—in a big distributed system there are always some parts that have failed. Hence, there are many more rare states, which is why a distributed system is harder to get right than a centralized one, in which many errors just reset the whole system to a known state.

In addition, a distributed system usually does not have a reset button and therefore needs a way to get from an arbitrary state to a good state (one that satisfies its invariant). This is called self-stabilization, first studied systematically by Dijkstra in a paper [Dijkstra 1974] that I and almost everyone else completely failed to understand; it's discussed in Lamport's chapter on concurrent algorithms (Chapter 4) and in Herman's chapter on self-stabilization (Chapter 5) in this volume. Much later it became clear that Dijkstra gave an example of a general strategy: have some state encoding a counter that measures how far away the system is from its invariant, with a way to decrement the counter repeatedly until it reaches zero. In Dijkstra's algorithm that counter is the number of adjacent processes that violate the invariant.

In an asynchronous distributed system, the processes can only communicate by sending messages to each other, and there's no way to know how long ago a received message was sent. Hence the only useful meaning a message conveys is a fact that is *stable*: once it is true, it remains true forever. For example, $x > 10$ is stable if x is a monotonic counter. A lease can imply "P holds until local time t," and if clocks are synchronized with skew Δ then "P holds until time $t - \Delta$" is stable. "P holds until Q" might be stable too, but a failure can make it impossible to evaluate Q, so this is best avoided. A fact about the state of an eventually consistent system (see Section 3.10) is stable only if it was true before the last sync operation. This stability principle puts a severe constraint on the design of distributed systems, and you can find many bugs by seeing where it is violated.

3.4 Atomicity and Commuting

The essential idea for taming the nondeterminism of concurrency is *atomicity*: if the actions of a process p commute with all the actions of other processes, then p has the same behavior as a process in which p's actions are grouped into larger *atomic* actions that behave as though each one runs sequentially—there is less nondeterminism, and it's easier to reason about the code.

An action a *right commutes* with another action c if in every reachable state the effect of doing a and then c is the same as the effect of doing c and then a: $a; c = c; a$. If a and b are sequential actions of process p in a system S, and a right commutes with *every* action c that can run in any other process of S, so that $a; c; b = c; a; b$, then S has the same behaviors as a system S' in which p has a single atomic action $a; b$. The action a is called a *right mover* because it moves to the right in the sequence of actions [Lipton 1975]. This works for any number of right movers $a_1, \ldots, a_n$ of p. Similarly, if b is a left mover (it left commutes with every c: $c; b = b; c$), then $a; c; b = a; b; c$. Thus, if x is an arbitrary p action and the b_i don't block, a sequence of p actions of the form $a_1; \ldots; a_n; x; b_1; \ldots; b_m$ (where the a_i are right movers and the b_j are left movers) has the same behavior as a single atomic p action that runs sequentially because any interleaved actions from other processes commute before or after the atomic action. Nondeterminism has been tamed.

When do two actions commute? They obviously do if they don't share any variables that one of them writes. So, actions on variables that are private to process p

are both left and right movers. There are more subtle cases of commuting actions (for example, two atomic increment operations), but they are not considered here.

This is not the whole story about atomicity because some variables, notably locks and message channels, have special properties that allow them to be shared safely even though their actions don't commute. If a and c are lock of the same lock, $a; c$ is impossible because a lock can't be locked twice without an intervening unlock. Similarly, if only process p does receive on a message channel q, a and c are actions on q, and a is receive, then $a; c$ is impossible if c is receive or send when q is empty, and commutes if c is send to a non-empty q. So, lock and receive are right movers, and similarly unlock and send are left movers. Informally, p receives information from other processes in the a phase (all right movers) and sends information to them in the b phase (all left movers).

Modern multiprocessors have relaxed consistency memory that buffers a processor's writes before making them visible to other processors [Adve and Gharachorloo 1996]. To keep this from breaking atomicity there must be a *fence* instruction as part of lock to ensure that the atomic block sees all preceding writes to shared variables, and another fence as part of unlock to ensure that all of the atomic block's writes are visible to all processes. The lock primitives should include these fences.

A good rule of thumb is the *scalable commutativity rule*: if the specifications of two actions commute, then it's possible to write code in which they run concurrently, which is important for keeping all the cores busy on modern CPUs [Clements et al. 2015]. For example, Posix file open returns the smallest unused file descriptor; if it returned an arbitrary unused descriptor, two opens could commute.

3.5 Taxonomy of Concurrency: Shard, Stream, or Struggle

It's helpful to classify the ways to program concurrency under five headings:

- *Really easy*: pure *sharding* or *streaming*. Either actions are *independent*, sharing no state except when you combine shards, or they communicate only by *producer–consumer* buffers. Either way, the actions are atomic: they behave as though they run sequentially.
- *Easy*: make a complex action *atomic* by locking all the variables or actions that it touches: the locks delay any actions that don't commute. The automatic version of this is called *transactions*.
- *Hard*: anything else that's serializable, usually with small atomic actions. With hard concurrency you can choose: do a formal *proof* (model checking might suffice) or have a *bug*. Stay away from it if you possibly can.

- *Eventual consistency*: all updates commute, so you get the same eventual result regardless of the order they are applied, but you have to tolerate data that is not up to date.
- *Nuisance*: actions can run concurrently but produce strange results if they do, and some convention avoids this. For example, opening a file while the directories on its path are being renamed can be surprising. The solution: don't do that.

The first three patterns make big actions atomic, and they provide *consistency* or serializability, which means that the concurrent system produces the same result as running the actions sequentially in some order. If there's an order that respects real-time (an action that starts after another action ends will be later in the order), consistency is also called linearizability.

Most applications use really easy concurrency because it doesn't require subtle reasoning and it's easy to check that the code obeys the rules for atomicity. Everything else is trickier; hence the slogan, "Shard, stream, or struggle." Easy concurrency is trickier because it's hard to check that the code is following the rules. Transactions in database systems do this automatically, and usually they also provide fault tolerance and handle deadlocks; the price is severe constraints on the code you can write. Hard concurrency is much trickier, and generally requires a proof. Eventual consistency is easy to code, but its clients must deal with a lot of nondeterminism.

The rest of this chapter discusses each of these patterns in turn.

3.6 Sharding

Sharding (also called striping or partitioning) is really easy concurrency that divides the state into pieces called shards that change *independently*. Only one process touches each shard, so the actions of processes that touch different shards don't depend on the interleaving; each process runs sequentially. A *key*, usually a name for the shard, determines which shard to use. Sharding is essential for scale-out, handling an arbitrarily heavy load by running on arbitrarily many machines. It lets you trade more resources (more bandwidth) for less latency. With enough shards the only limit to this is the budget, as cloud services for search, email, and so on show. With sharding there are no locks or conventions about owning variables.

The simplest example is disk striping: a few bits of the address are the key that chooses the disk to store a given block, and all the disks read or write in parallel. Fancier is a sharded key–value store with ordered keys; $n - 1$ *pivot* values divide the keys into n roughly equal chunks. To look up a key, use the pivot table to find its shard. Everything is trivially concurrent except for rebalancing the shards or adding shards, which is trickier.

Another easy example is independent clients of a service for which all the shared state is read-only: viewing content such as newspapers or videos, shopping, or email servers. As with the key–value store, optimizing the layout and caching read-only state is trickier.

A cloud's many independent clients make it easy to shard both computing and storage. A compute shard is either a virtual machine or a "serverless function," a short-lived chunk of computation that runs with some fixed OS, language-specific runtime, and initial state, but can read and update state in storage objects. A storage shard, called an object or blob, is a key–value pair where the value is just a sequence of bytes, possibly a few terabytes long. Clients can use objects to store sets of database rows or other structures. As usual, bigger shards perform better because there's a minimum cost to run a virtual machine or access an object (and cloud providers throttle clients that access objects too fast), but the client has to manage the batching, which adds complexity. Of course, there is trickier concurrency in the code for scheduling clients and laying out storage.

The sharding can be flat (usually by hashing the key to find the shard as in distributed hash tables [Stoica et al. 2003], or hierarchical if there are natural groupings or subsets of keys, as with path names for files, DNS names, and Internet addresses. Hierarchy is good for change notifications since it makes it easy to notify all the "containing" shards that might need to know, but not so good for load-balancing since there may be hotspots that get much more than their share of traffic. To get around this, make the path names into ordered keys, disregarding their structure and treating them as strings, and pivot on them as above.

Often, there's a *combining function* for results from several shards. A simple example is sampling, which just takes the union of a small subset from each shard; the union is serial, but it's typically processing log n out of n items so this doesn't matter. Union generalizes to any linear (homomorphic) function between two monoids $f : M \longrightarrow N$. This means that f preserves operations so that $f(a +_M b) = f(a) +_N f(b)$, and $f(0_M) = 0_N$. Each shard evaluates a linear function, and then a tree combines the results with $+_N$ [Budiu et al. 2016]. The familiar map and reduce operators of map-reduce [Dean and Ghemawat 2008] are linear.

A tricky example is sorting on n processors: shard the input arbitrarily, sort each shard, and combine by merging the sorted shards. A straightforward merge requires processing all the data in one place, and Amdahl's Law will limit the performance. To avoid this,

- sample the data in parallel to make a pivot table;
- send the data to the shards in parallel—each item in shard i precedes any item in shard $i+1$; and
- sort the data in parallel in each shard.

Concatenating the data from the n shards in order, rather than merging it, yields a sorted result [Arpaci-Dusseau et al. 1997].

The independent shards sometimes have to come back together. This means combining the outputs, usually using a streaming tree structure. Fancy versions of this are called *fusion*. Performance depends on minimizing communication, which can't be faster than the speed of light, so the latency to access n bits grows at least like $\sqrt[3]{n}$ in our 3D world. Put another way, fast computing needs locality, and it's easier to make small things local, so saving space can also save time. To get locality as a system scales up, the shards have to be almost independent.

Simple versions of combining often just put shards into a single federated name space by making a new root with all of them as children.

- In a file system this is called *mounting*, and the shards stay independent.
- In a source code control system, the shards are *branches* and synchronization is *merging*.
- Modules that satisfy the same specification can federate even if they use different techniques, as in SQL query processing, numerical computing, or mixtures of experts in machine learning.

3.7 Streaming

Streaming (or pipelining) is the other really easy kind of concurrency:

- Divide the work for a single item into k stages, each one of which runs sequentially.
- Put one stage on each processor.
- Pass work items along the chain as messages, perhaps with buffering between the stages.

A stage runs sequentially because it shares no variables with other stages except for the message channels, so that the sequence "receive item; process; send item" is atomic. If each stage takes the same amount of time it's systolic, which is even better because it's easier to optimize the scheduling. This scheme generalizes to *dataflow*, where the work flows through a DAG. The number of distinct processing stages limits concurrency. Batching work reduces the per-item overhead.

You can evaluate any expression that doesn't have side effects this way because it forms a tree, or a DAG if there are common sub-expressions. An important example is a database query, where the primitive operations are expensive enough that the overhead of streaming is negligible.

Map-reduce [Dean and Ghemawat 2008] combines sharding and streaming, alternating a sharded map phase with a combining reduce phase that also redistributes the data into shards that are good for the next phase. The redistribution is the reason for having phases rather than a single DAG. You can reuse the same machines for each phase or stream the data through a DAG of machines. The combining phase of map-reduce illustrates that even when there is a lot of independence, communication is often the bottleneck.

3.8 Easy Concurrency

The idea of easy concurrency is that a process should exclusively *own* any variable that it touches. Locks are the most common way to temporarily give a process ownership of some variables:

- Every shared variable is protected by a lock.
- A process must lock it before touching the variable.
- No other process can lock it until the holder unlocks it.

Thus the locks provide mutual exclusion: if one process is holding a lock, others are excluded. The part of the program where the lock is held is sometimes called a *critical section*. The sequence of actions between the first lock and the first unlock is atomic. There are some complications, discussed below.

Dijkstra invented these ideas, using the terms "mutual exclusion" and "critical section." He devised a construct called a *semaphore* [Dijkstra 1968a], an integer s with two atomic operations associated with it:

- $s.P$ (lock/wait): **until** $s > 0$ **do** block process **end**; $s := s - 1$
- $s.V$ (unlock/signal): $s := s + 1$

If s is initially 1, $s.P$ does the job of lock and $s.V$ of unlock. If s is initially 0, $s.P$ does the job of wait and $s.V$ of signal. As Dijkstra pointed out in an appendix [Dijkstra 1968b], these are two very different uses of semaphores. It's tricky to code atomic P and V using only atomic memory read and write, and since the 1970s machines have had test-and-set instructions that provide the necessary atomicity in the hardware.

More recently compare-and-swap instructions do this job, and also make wait-free synchronization possible [Herlihy 1991].

Ownership can also be by programming convention. A process owns its private variables permanently; and if there's only one external reference r to a data structure, a process may own all its variables by doing

$$l.lock;\ myR := r;\ r := \text{nil};\ l.unlock$$

and give up ownership with $r := myR$. *Epochs* are another ownership convention, described below. It's usually hard to check that such conventions are obeyed.

Owning actions rather than variables is a more general way to ensure that a process' actions commute with actions of other processes. To do this, group the actions into sets such that two actions are in the same set if they don't commute (and hence break atomicity if they run concurrently), such as reads and writes of the same variable. Have a *lock* variable that protects each set, with the rules that:

- Before running an action, a process must hold the action's lock.
- Two locks *conflict* if their actions don't commute. For example, two reads commute, but writes of the same variable don't commute with reads or other writes. The abstraction of this is exclusive versus shared locks.
- A process must wait for a lock if another process holds a conflicting lock.

These locking rules ensure that any concurrent action c that happens between an action a and the action r_a that releases a's lock commutes with a, since an action that doesn't commute must wait for its conflicting lock. So as we saw earlier, $a; c; b; r_a$ behaves the same as $c; a; b; r_a$, where b is the rest of the work that a's process is doing, and hence neither a's process nor c's process sees any concurrency until r_a releases the lock. The atomic actions in the concurrent processes are *serialized*. Another way of saying this is that each atomic action is equivalent to a single action that occurs at a single instant called its *commit point*. It's valid to reason about the code of such an atomic action as though it runs sequentially, with nothing going on concurrently.

The way to reason about more than one atomic action is to define a *lock invariant* on the state for each lock; the invariant should hold except perhaps inside the code of an atomic action that holds the lock. This means that you can assume the invariant when locking and must establish it before unlocking. You can think of the lock as owning its variables and handing them off temporarily to a process that locks it. Another way of saying this: choose atomic actions wisely. For example, to increment x atomically it's not enough to hold a read lock when fetching x and a write lock when storing it; the lock must cover the whole atomic sequence. Otherwise two concurrent executions of $x := x + 1$ can increment x by either 1 or 2.

There are three tricky things about easy concurrency: enforcing the lock discipline, reasoning about the state in between atomic actions, and dealing with deadlock.

Discipline: Before running an action, you must hold its lock. Unfortunately, locking is hard to debug because the program will still work most of the time even if it runs actions without having locked them. Tools like Eraser [Savage et al. 1997] can detect most of these errors by checking that the same lock is held whenever the program touches a variable.

Lock invariant: When you lock you can assume nothing about the state the lock protects except the lock invariant. Unless you are proving the correctness of the program, there is not much you can do to check this. It helps to write down the invariant.

Deadlock: Deadlock is a cycle of processes, each waiting for the next to release a lock. Detecting deadlock is expensive; the usual way to avoid it is to define a partial order on the locks, and only acquire a lock if it's bigger than all the ones you already hold. This prevents cycles. Dijkstra defined deadlock and gave this solution in Dijkstra [1968a]; he called it a "deadly embrace."

Transaction processing systems automate easy concurrency. They solve all these problems in draconian fashion, making each transaction atomic in spite of concurrency (and even when there are crashes, by writing all changes to a persistent log, logging a *commit record* when the transaction commits, and replaying the log after a crash if the transaction has a commit record in the log). The only thing the programmer has to do is to delimit each atomic transaction with begin and commit actions.

- They interpose on the client's reads and writes of the shared state to acquire the necessary locks.
- They don't allow a client to keep *any* private state after a transaction commits; the client has to reread everything from the shared state. This is essential for the automatic locking to work, and it also makes it easy to update the transaction's code even if it's running—the current transactions keep running with the old code, and any new ones start with the new code.
- They detect deadlock and abort one of the transactions involved, undoing any state changes.
- They don't release locks until the transaction commits and any changes have been written to persistent storage.

As a result, transactions are the pixie dust of computing. They take an application that understands nothing about fault tolerance, concurrency, undo, persistent storage, updating or load-balancing, and magically make it atomic, abortable, immune to crashes, easy to update, and easy to distribute across a cluster of machines. Of course, these benefits have a cost, both in execution time and in flexibility.

A lock is a resource, and like any resource it can become a bottleneck, so it needs to be instrumented. DEC built the first computing clusters, in which you could do more computing simply by adding machines. But one large cluster didn't scale, even though no physical resource was a bottleneck. It turned out there was a single lock that was 100% busy, but it was a very hard to figure this out because there was no way to measure lock utilization.

An important special case of easy concurrency is *epochs*, a batching technique that maintains some invariant on the state except at the end of an epoch. This is a special case of locking that owns certain actions throughout the epoch so that they can only occur when the epoch ends. The code follows these rules by convention; there's no lock variable that's locked and unlocked, so epochs are cheap but easily subverted. Most often the action that is locked is deleting an object, so that objects won't disappear unexpectedly; instead of actually doing a deletion during the epoch, you queue it to happen at the end. This is easy because it doesn't depend on any other state, but it has to be okay to defer the deletions, as it is in wait-free code, which changes an object by creating a new version and must delete the old version.

Sometimes the global lock prevents *any* changes to certain objects, keeping them immutable during the epoch; you update a copy of the object and delete the original at the end of the epoch. Read-copy-update uses this strategy to avoid a lot of explicit locking [McKenney and Slingwine 1998].

Another way to make big actions atomic is *optimistic concurrency control* (OCC). The idea is to let the code of an atomic action touch variables fearlessly but keep any changes in a write buffer private to the code. When the action commits, check that any variables that were read have not been written by someone else, usually by keeping a version number on each variable. If the check fails, abort: discard the changes and *retry* the action. If it succeeds, commit and install the changes. This check and install itself needs to be made atomic by locking, but it can be very fast and hence can be protected with a single global lock. The commit point is when this lock is held. Unfortunately, you might see inconsistent data, but if you do you will always abort. Hardware transactional memory works this way, using the cache for the write buffer. OCC works well when conflicts are rare since you only pay when there is a conflict. Locking is better when conflicts are common since waiting is better than the wasted work of many aborts and retries. OCC becomes less efficient as load increases, and under high load its performance can totally collapse.

There are many constructs other than locks and conditions for coding easy concurrency. Each one handles some problems very elegantly, but unfortunately falls apart on others. My view is that there's only a limited amount of elegance to go around; don't spend it all in one place. Locks are an inelegant but all-purpose tool for atomicity; conditions do the same for scheduling.

3.8.1 Monitors

A monitor is a way to make it easier and more reliable to program easy concurrency by packaging some shared data together with the code that operates on it [Hoare 1974, Lampson and Redell 1980]. The code takes the form of procedures or methods that the monitor makes atomic by automatically acquiring a lock on entry and unlocking it on exit or wait. If every shared data item is a declared variable of the monitor, it's easy to statically check the rules for easy concurrency. Static checking is harder if you follow a pointer from a monitor variable to reach an item, but there's a simple rule: such pointers shouldn't leave the monitor. Monitor procedures are like transactions that are not fault-tolerant, with the additional feature that all the code that touches the monitor's variables is gathered together in one module or object.

Monitors lead naturally to a program that has a dual message-oriented form, with a monitor that does work for threads replaced by a process that handles messages [Lauer and Needham 1978]. The former is more flexible, but the latter is better if processes are expensive and typically don't share memory.

Processes and monitors	*Messages*
Process	message (I/O control block)
Monitor	process (resource manager)
entry procedure a	message port a
	SendMsg(a, $args$) ... GetReply
call $a(args)$	immediate reply
$p :=$ **fork** $a(args) \ldots$ **join** p	delayed reply
return from a	SendReply to a
monitor	process main loop:
	$m :=$ WaitForMsg; **case** m **of**
proc ep_1 $\{\ldots\}$;	ep_1: $\{\ldots\}$;
$\ldots$;	$\ldots$;
proc ep_n $\{\ldots\}$	ep_n: $\{\ldots\}$
conditions, wait/signal	wait for selected messages

This duality is not general; it depends on writing the monitor program in a stylized way. It does show, however, that the many different ways of writing a concurrent program conceal a smaller number of logical structures.

Monitors evolved from Dijkstra's notion of *secretaries*. He envisioned a secretary as a separate process that client processes could call on by sending a single message and waiting for a reply, an interesting hybrid of monitors and messages [Dijkstra 1971]. The dual message pattern evolved into Hoare's [1978] communicating sequential processes.

3.8.2 Scheduling

Often, one "consumer" process needs to wait for another "producer" process to do something. The simplest case is waiting for a lock to be free or a message to arrive. More complex cases come up when there are resources that need to be multiplexed between processes, perhaps based on priorities or on the need to have several resources at the same time. In principle, a construct of the form **await** *b* can meet all these needs, where *b* is an arbitrary Boolean expression. The consumer simply evaluates *b* repeatedly until it becomes true; if *b* needs to hold a lock *l*, use busy waiting code of the form

l.lock; **until** *b* **do** *l.unlock*; *l.lock* **end**; do work; *l.unlock*.

This is called a spin lock. Unfortunately, it keeps a CPU busy during the wait. *Polling* alleviates this problem by adding a delay that gives up the CPU between *l.unlock* and *l.lock* in the loop. This is simple, but it's efficient only if you can tolerate enough latency (that is, poll infrequently enough) that you usually find work to do.

A more practical alternative is to wait on a *condition variable cv* in the loop:

l.lock; **until** *b* **do** *cv.wait*(*l*) **end**; do work; *l.unlock*.

Any producer that makes *b* true must do *cv.signal*. The *cv.wait*(*l*) atomically unlocks *l* (so that the producer can lock it) and blocks the consumer process until there's a signal on *cv*, and then locks *l*. Extra signals are harmless because this code doesn't assume that *b* is true after a wait. However, a producer bug that omits a signal causes an unbounded wait. One way to recover is to add a harmless extra signal every so often; now you have a somewhat ugly hybrid of conditions and polling.

A reverse hybrid polls for a while, hoping that *b* will become true soon, and then waits when the hope is frustrated. This is the usual code for a lock that is expected to be held for a short time on a multiprocessor.

It's important for the body of the loop to be atomic because in the non-atomic sequence

[C] test b; [P] make b true; *cv.signal*; [C] *cv.wait*

the signal will be lost. The alternative is for *cv.signal* to set a "wakeup-waiting switch" in *cv* that makes the next *cv.wait* clear the switch and continue immediately. Dijkstra's design, using a semaphore instead of *cv*, doesn't have this problem because the semaphore counts *signal*s. He thought that the program should exactly pair *signal*s and *wait*s.

The original design for monitors gives control to the consumer immediately without giving up the lock and requires the producer to establish b so that the consumer can rely on it. This avoids the extra test for b in the consumer, which is usually cheap, at the price of extra context switches and extra care in the producer.

When there is a lot of contention for a resource, the program often needs more control over scheduling than just waking up the process that has waited longest—perhaps one process has higher priority, or needs special paper loaded in a printer, or should run at the same time as another process. A condition for each class of consumers or even for each consumer can provide this, together with custom code for whatever scheduling discipline you want. A universal way of handling contention is *exponential backoff*: a process responds to some overload signal by decreasing its offered load (rate) by some factor. Examples: Ethernet, Internet TCP, Wi-Fi, spin locks that wait before retrying.

An I/O device cannot strictly follow the producer–consumer pattern and cannot hold a lock. Instead, it has a partner CPU process that can do these things, and the device interrupts a CPU to run code that signals the partner.

3.9 Hard Concurrency

If you write a concurrent program in which a process doesn't own all the variables it touches, either because they are locked or by some programming convention, you are doing hard concurrency, and unless you are willing to have nondeterministic bugs you need a proof. This means you should learn how to use a formal system like TLA+ [Lamport 2003]. See Chapter 4 by Leslie Lamport. There are many examples of hard concurrency in widely deployed systems, and most of them do have bugs. Prudent use encapsulates the hard concurrency in routines with simple atomic specifications; typical examples are mutual exclusion (lock and unlock), distributed fault-tolerant consensus, and file systems.

A proof of hard concurrency depends on an *inductive invariant* I; a state predicate that is initially true and strong enough that every action maintains it: $\forall a, s, s' : I(s) \wedge a(s, s') \implies I(s')$. Coming up with I is harder than you might think. Dijkstra preferred to reason constructively, building up the program and the proof in parallel; he used invariant arguments somewhat informally in developing a concurrent garbage collector [Dijkstra et al. 1978].

3.9.1 Multiple Versions and Wait-free Concurrency

Multi-version concurrency control (MVCC) is a third way to code atomicity. It remembers the state after every atomic action. An action can use its start time as its commit point since it can always read the state as of that time. However, to write a variable that has already been read by a later action it must abort and move its commit point after the read; this works well for actions that only write private variables, most often after checking some property of the state such as whether the bank's books balance. Each version is immutable, so a cache entry tagged with the version it applies to is never invalid. MVCC is the basis for a large example of hard concurrency: a full-blown wait-free caching transaction system [Levandoski et al. 2015].

Multiple versions are also the basis for the general form of wait-free (nonblocking) computation: make a new version and then splice it in with a compare-and-swap instruction, retrying if necessary [Herlihy 1991]. Wait-free is good because a slow process holding a lock can't force others to wait. If there is contention, retry can *livelock* (keep retrying); the fix is for a conflicting process to *help* with a failing update.

3.10 Eventual Consistency

The specification for eventual consistency (EC) in a system that replicates data to a set of nodes is that a read sees an *arbitrary subset* of all the updates that have been done [Birrell et al. 1982]. This is easy to code:

- Make updates commute.
- Broadcast updates to all nodes.

The simple case of commuting updates is a blind write $v :=$ constant to a variable v, perhaps in a hierarchical name space. To make two writes to v commute, timestamp them and let the last writer win. Collapse the update history into just keeping the timestamp of the last write with each variable, and when a write arrives, apply it if its timestamp is later. A deletion must leave a time-stamped tombstone. If you keep the whole update history, timestamps make arbitrary updates commute, but that's expensive.

To broadcast, arrange the nodes in a connected graph; the simplest is a ring. When a node receives an update for the first time, it sends it on to all of its neighbors. There are many variations. Usually there's a sync operation, which guarantees that after it ends every read it sees all the updates that precede the start of the sync.

EC is simple and highly available since you can always run using only local data. The clients pay the piper: they must deal with stale data. Three of many examples are name services like DNS (which has no sync), key–value stores like Dynamo [DeCandia et al. 2007], and "relaxed consistency" multiprocessor memory, in which sync is called "fence" or "barrier" [Adve and Gharachorloo 1996]. This design works well for a shopping cart, but registering a new name needs to be atomic. A simple solution is to register each name at a single master node for the name; DNS works this way, and you are out of luck if the master is down. For high availability, use a consensus protocol to replicate the master.

In most file systems, writes are buffered, and this means that there's no guarantee that data is persistent until after an fsync. After a failure this is somewhat similar to EC: there may be updates that have been read but that no one will ever see again.

If an interactive system has actions with highly variable latency, such as reading something over a network, it should plunge ahead rather than keep the user waiting for a response. Usually the response affects the state, so you need a way to postpone that effect. This requires being very clear about the semantics of unordered sets of operations, a form of EC. Web pages do this all the time, and word processors often do it for slow operations like hyphenation.

3.11 Nuisance Concurrency

When actions can run concurrently but produce strange results if they do, use some higher-level convention to avoid this. A familiar example is opening and renaming files. If the directories involved are themselves being renamed many strange things can happen, since a file system usually doesn't hold locks to serialize these actions. Applications use conventions to avoid depending on directories that might be rearranged while the app is running. A specification for nuisance concurrency (which you don't normally care about because you're avoiding it) has the same flavor as EC: it collects all the changes that can occur during a non-atomic action, and the specification for the action chooses a subset of these changes nondeterministically to make the state that it acts on. The difference is that the client is not expecting the inconsistency.

EC and nuisance concurrency are examples of a general strategy: move the nondeterminism of interleaved actions to nondeterminism in the specification, where it's easier to convince yourself that you can tolerate it.

3.12 Conclusion

Concurrent programming has become essential to computing in practice because although transistors are no longer getting faster they are still getting cheaper, and hence sequential programs are no longer getting faster but CPUs are still getting cheaper. The ideas of sequential process, mutual exclusion, and efficient waiting for a resource, introduced by Dijkstra in the 1960s, are still its foundations. The main change is that their easiest forms, sharding and streaming, are now the most widely used.

References

S. V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: A tutorial. *IEEE Comput.* 29, 12, 66–76. DOI: https://doi.org/10.1109/2.546611.

A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. 1997. High-performance sorting on networks of workstations. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1997).* ACM Press, 243–254. DOI: https://doi.org/10.1145/253260.253322.

A. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. 1982. Grapevine: An exercise in distributed computing. *Commun. ACM* 25, 4, 260–274. DOI: https://doi.org/10.1145/358468.358487.

M. Budiu, R. Isaacs, D. Murray, G. D. Plotkin, P. Barham, S. Al-Kiswany, Y. Boshmaf, Q. Luo, and A. Andoni. 2016. Interacting with large distributed datasets using Sketch. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV 2016)*. Eurographics Association, 31–43.

A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. 2015. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.* 32, 4, 10:1–10:47. DOI: https://doi.org/10.1145/2699681.

J. Dean and S. Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1, 107–113. DOI: https://doi.org/10.1145/1327452.1327492.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2007).* ACM, 205–220. DOI: https://doi.org/10.1145/1323293.1294281.

E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9, 569. DOI: https://doi.org/10.1145/365559.365617.

E. W. Dijkstra. 1968a. Cooperating sequential processes. In F. Genuys (Ed.), *Programming Languages: NATO Advanced Study Institute*. Academic Press, 43–112. Reprinted in P. Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming*. Springer, 65–138. DOI: https://doi.org/10.1007/978-1-4757-3472-0_2.

E. W. Dijkstra. 1968b. The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5, 341–346. Reprinted in P. Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming*. Springer, 139–152. DOI: https://doi.org/10.1007/978-1-4757-3472-0_3.

E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inform*. 1, 115–138. Reprinted in P. Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming.* Springer, 198–227. DOI: https://doi.org/10.1007/978-1-4757-3472-0_5.

E. W. Dijkstra. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11, 643–644. DOI: https://doi.org/10.1145/361179.361202.

E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11, 966–975. DOI: https://doi.org/10.1145/359642.359655.

M. Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1, 124–149. DOI: https://doi.org/10.1145/114005.102808.

C. A. R. Hoare. 1974. Monitors: An operating system structuring concept. *Commun. ACM* 17, 10, 549–557. DOI: https://doi.org/10.1145/355620.361161.

C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8, 666–677. DOI: https://doi.org/10.1145/359576.359585.

L. Lamport. 2003. *Specifying Systems*. Addison-Wesley, Boston.

B. W. Lampson and D. D. Redell. 1980. Experience with processes and monitors in Mesa. *Commun. ACM* 23, 2, 105–117. DOI: https://doi.org/10.1145/358818.358824.

H. C. Lauer and R. M. Needham. 1978. On the duality of operating system structures. In *Proceedings of the 2nd International Symposium on Operating Systems*. Reprinted in 1979. *Operating Systems Review*, 13, 2, 3–19. DOI: https://doi.org/10.1145/850657.850658.

C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. S. Sanchez, and T. B. Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495, 1079. DOI: https://doi.org/10.1126/science.aam9744.

J. J. Levandoski, D. B. Lomet, S. Sengupta, R. Stutsman, and R. Wang. 2015. High performance transactions in Deuteronomy. In *7th Biennial Conference on Innovative Data Systems Research (CIDR 2015)*. www.cidrdb.org.

R. J. Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18, 12, 717–721. DOI: https://doi.org/10.1145/361227.361234.

P. E. McKenney and J. D. Slingwine. 1998. Read-Copy-Update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 1998)*. 509–518.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4, 391–411. DOI: https://doi.org/10.1145/265924.265927.

I. Stoica, R. T. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Netw.* 11, 1, 17–32. DOI: https://doi.org/10.1109/TNET.2002.808407.

4 Concurrent Algorithms

Leslie Lamport

4.1 Introduction

A concurrent algorithm is one in which concurrently executing, independent processes interact with one another. A parallel algorithm is one in which the processes are not independent but were created to speed up execution of a single computation. There is no precise distinction between parallel and concurrent algorithms, and a concurrent algorithm may be used to synchronize the separate processes executing a parallel algorithm. However, they differ in practice and pose different problems. Distributed algorithms are a class of concurrent algorithms.

Dijkstra published five influential papers about concurrent algorithms. The first was the most important because it began the study of concurrent algorithms. This chapter describes most of the algorithms in those five papers, along with background material for some of them. Their descriptions here do not reflect how the algorithms were presented in the papers. I have extensively modified Dijkstra's notation and the way he described the algorithms, reflecting what has been learned in the decades since he did that work (and perhaps my biases). In particular, algorithms are written in the sort of informal pseudo-code I might have used in the 1980s and is still widely used today.

Dijkstra was careful to acknowledge colleagues who influenced his papers, often including them as coauthors. I believe he was the driving force behind these five papers, and I am reasonably sure that he did all the actual writing himself. I do not mean to belittle any of the coauthors of these papers (especially since I am one of them). However, for brevity, the papers are referred to simply as Dijkstra's.

4.2 Mutual Exclusion

In 1965, Dijkstra published in *Communications of the ACM* (CACM) a paper [Dijkstra 1965] containing what I believe was the first concurrent algorithm to appear in print. More important than the paper's algorithm was its introduction of the mutual exclusion problem that the algorithm solved. Even more important than that, the paper introduced the way we now think about concurrently executing processes. And it did all this in one page.

In the mutual exclusion problem, there are N processes, each of which may repeatedly execute a section of code called its *critical section*. The remainder of its code is called the *noncritical section*. A process may halt only in its noncritical section. The problem is to add synchronization code to ensure that executions of the critical section by two different processes cannot occur at the same time. Dijkstra discussed this problem in EWD35, apparently written in 1962. He wrote there that, in 1959, the mathematician Theodorus Jozef Dekker found a solution for two processes, and that:

> [F]or almost three years, this solution has been considered a "curiosity", until these issues at the beginning of 1962 suddenly became relevant for me again...[1]

It was building the THE operating system [Dijkstra 1968] that made the problem relevant. That system was implemented as a collection of separate processes. Mutual exclusion was used, for example, to prevent two different processes from trying to print with the same printer at the same time.

4.2.1 Preliminaries

Dekker's and Dijkstra's algorithms, as well as many later mutual exclusion algorithms, are based on what I call the *one-bit protocol*: Each process has a flag (a Boolean-valued variable) that initially equals **true**; it enters the critical section by setting its flag to **false** and then waiting until it has read the value **true** in every other process's flag.[2] A process resets its flag to **true** upon exiting the critical section. No two processes can be executing their critical sections at the same time because the last one to set its flag to **false** would have read the other's flag equal to **false** if that other process were still in its critical section.

The one-bit protocol ensures mutual exclusion, but it does not solve the mutual exclusion problem. (I call it a protocol because it is used only as part of a complete algorithm.) If two processes both set their flags to **false** before reading the other process's flag, then something must be done to prevent both of them from waiting forever for the other to set its flag to **true**.

[1] From a translation by Martien van der Burgt and Heather Lawrence on the `cs.utexas.edu` web site.

[2] Following Dijkstra, I am using a flag whose value indicates if the process *does not* want to enter the critical section. These days, the protocol is described with a flag having the opposite meaning.

```
variable $flag[i \in 1..N] = $ true
process $i \in 1..N$ do
  while true
    do  noncritical section ;
      $L$: ⟨$flag[i] :=$ false⟩ ;
        for $j \in 1..(i-1)$
          do if ⟨$\neg flag[j]$⟩ then ⟨$flag[i] :=$ true⟩ ;
                                     ⟨await $flag[j]$⟩ ;
                                     goto $L$
             fi
          od ;
        for $j \in (i+1)..N$
          do ⟨await $flag[j]$⟩ od ;
        critical section ;
        ⟨$flag[i] :=$ true⟩
    od
od
```

Figure 4.1 The one-bit algorithm.

In Dekker's and Dijkstra's algorithms, this deadlock is broken by having one of the processes set its flag to **true** to let the other process enter the critical section before it does. This is also done in a simpler way by what I call the *one-bit algorithm*. This algorithm, for *N* processes numbered 1 through *N*, is described in Figure 4.1 using the following notation. The **variable** statement declares the shared variables, which can be accessed by any process, and their initial values. The expression *i*. .*j* equals the set of all integers *k* with $i \leq k \leq j$. Variable *flag* is declared to be an array with index set 1. .*N* such that *flag*[*i*] equals **true** for all *i* in that set. The statement

process *i* ∈ 1..*N* **do** . . . **od**

declares that there is a process for each number in 1. .*N*, where "..." is the code for process number *i*. The meanings of the **while**...**do**...**od**, **if**...**then**...**fi**, and **goto** statements[3] should be obvious; and := denotes assignment. Boolean negation is written ¬. The **for** and **await** statements and the angle brackets ⟨⟩ require explanation.

Execution of the statement

for *j* ∈ *S* **do** *T* **od**

means executing *T* once for each value of *j* in the set *S*. The executions for different values of *j* can occur in any order. At that time, Dijkstra would use a conventional **for**

[3] In 1965, Dijkstra had not yet begun his crusade against the **goto**, and he used it freely. Simplicity is crucial for avoiding errors in concurrent algorithms. A **goto** should be used if it simplifies the algorithm's description.

statement or other looping construct to perform the executions of statement T in a specified order, even when the order of execution didn't matter. He would sometimes mention that the order was irrelevant, but often didn't—perhaps because he expected it to be obvious to the reader.

Dijkstra assumed that the execution of a concurrent algorithm can be described as a sequence of atomic actions, each performed by a single process. (I will usually eliminate the word "atomic".) Exactly how execution of an operation is broken down into separate actions doesn't matter for single-process algorithms. It does for concurrent algorithms. Suppose two processes concurrently execute the statement $x := x + 1$. This increments x by 2 if execution of the statement is a single action. There are many other possible outcomes if reading and setting each bit in the representation of x is a separate action. Writing $\langle\, x := x+1 \,\rangle$ indicates that the execution is a single action. For an expression E, writing $\langle\, E \,\rangle$ indicates that evaluation of E is a single action. Execution of an operation not in angle brackets can be split in any way into a sequence of actions.

The statement $\langle$**await** $E\rangle$ can be interpreted in two ways. One way is that the statement is executed only when the expression E equals **true**, and its execution just causes control to pass to the following statement. The other way is that it is equivalent to:

$$L \,:\, \langle\ \textbf{if}\, \neg E\ \textbf{then goto}\ L\ \textbf{fi}\ \rangle,$$

which is the way Dijkstra wrote it (without the angle brackets). With the first interpretation, its execution consists of a single action. The second interpretation allows that action to be preceded by a sequence of actions that have no effect. An action that has no effect is unobservable, so those two interpretations are equivalent.

Let us now examine the algorithm. To enter its critical section, process i must set $\mathit{flag}[i]$ to **false** and then execute the two **for** loops. Completing execution of those loops requires reading $\mathit{flag}[j]$ equal to **true** for every other process j. The algorithm therefore implements the one-bit protocol for entering the critical section, ensuring mutual exclusion.

The algorithm also satisfies a property called *deadlock freedom*[4]: whenever some process is trying to enter its critical section, eventually (then or at a later time) some process is in its critical section. The proof is by contradiction. We assume

[4] This name is misleading, because deadlock usually means that processes are waiting forever at **await** statements. Here, deadlock freedom also rules out livelock, where processes keep executing statements but never make progress.

that some process is trying to enter its critical section but no process is ever in its critical section, and we obtain a contradiction.

(1) Eventually, every process is forever either in its noncritical section, or waiting at an **await** statement, or looping through statement *L*.

Proof. By the code and the assumption that no process ever again enters the critical section. ■

(2) Eventually, every process is forever either in its noncritical section or waiting at an **await** statement.

Proof. Since no process is ever in its critical section, we need only show that no process is forever looping through statement *L*. If there is such a process, let process *i* be the lowest-numbered one. The code implies process *i* never examines the *flag* of any other process looping through statement *L*. By step 1, all the *flag* values it examines eventually never change, so it must eventually either exit its first **for** loop or wait forever at an **await** inside it. Hence, eventually no more processes can be looping through statement *L*. ■

(3) Let process *i* be the lowest-numbered process with *flag*[*i*] eventually always equal to **false**.

Proof. Such an *i* exists by step 2 and the assumption that some process is waiting to enter its critical section. ■

(4) Contradiction.

Proof. By step 3, process *i* must be waiting forever for *flag*[*j*] to equal **true**, for $j < i$. This implies *i* is in its first **for** loop, so *flag*[*i*] equals **true**, contradicting step 3. ■

This algorithm was never mentioned by Dijkstra. It was discovered independently by James E. Burns, me, and perhaps others in the 1970s. To my knowledge, it was not published until the 1980s [Burns et al. 1982, Lamport 1986]. However, it is so simple that it's hard to believe that Dijkstra, who understood the one-bit protocol, had not already discovered it by 1965. He might have, but then dismissed it for a reason discussed below.

4.2.2 Dekker's Algorithm

Deadlock freedom means that if some process *i* is trying to enter the critical section, then some process *j* eventually enters it. It does not imply that *i* eventually enters it. The one-bit algorithm gives lower-numbered processes priority. A process may try forever to enter the critical section while lower-numbered processes

```
variables $flag[i \in \{0,1\}] = \textbf{true},\ turn \in \{0,1\}$
process $i \in \{0,1\}$ do
  while true
    do    noncritical section ;
       $L1$: $\langle flag[i] := \textbf{false} \rangle$ ;
       $L2$: if $\langle \neg flag[1-i] \rangle$
                then if $\langle turn = i \rangle$ then goto $L2$
                                                  else  $\langle flag[i] := \textbf{true} \rangle$ ;
                                                        $\langle \textbf{await}\ turn = i \rangle$ ;
                                                        goto $L1$
                     fi
             fi ;
             critical section ;
             $\langle turn := 1 - i \rangle$ ;
             $\langle flag[i] := \textbf{true} \rangle$
    od
od
```

Figure 4.2 Dekker's algorithm.

keep entering and leaving. One might like the stronger property that every trying process eventually enters the critical section. For a reason that should be obvious later, this property has come to be called *starvation freedom*.

Dekker's algorithm for two processes achieves starvation freedom by using a dynamic priority rather than the fixed priority based on process number of the one-bit algorithm. It has an additional variable *turn* whose value indicates which process has priority. If *turn* equals *i*, then process *i* waits with *flag*[*i*] equal to **false**; otherwise, it waits with *flag*[*i*] equal to **true**. Upon exiting the critical section, a process sets *turn* to the number of the other process. The algorithm, as it appeared in EWD123 (but with more modern notation), is in Figure 4.2. The processes are numbered 0 and 1, and the declaration of the variable *turn* indicates that its initial value can be either 0 or 1.

In the algorithm, processes execute the one-bit protocol to enter the critical section, guaranteeing mutual exclusion. Here is Dijkstra's proof of deadlock freedom, restated rather tersely in terms of our notation.

> Suppose both processes are trying to enter the critical section. If neither succeeds, then the value of turn remains constant. It is then easy to see that process number turn must eventually wait with *flag*[*turn*] equal to **false** while process 1 − *turn* waits with *flag*[1 − *turn*] equal to **true**, allowing process turn to enter the critical section. If only a single process i is trying to enter, then process 1 − *i* must reach the noncritical section with *flag*[1 − *i*] equal to **true**, allowing process i to reach the critical section.

This argument is incomplete. It fails to consider the possibility that process *i* reads *flag*[$1 - i$] equal to **false** in the outer **if** test and then waits forever with *turn* equal to $1 - i$ while process $1 - i$ remains forever in its noncritical section. It's not difficult to show that this can't happen, but the proof is not trivial.

Dijkstra's proof was based on considering all possible behaviors of the algorithm. The lacuna in the proof illustrates that this kind of reasoning is dangerous; it is hard to think of all the possible behaviors of a concurrent algorithm. While Dekker's algorithm is correct, we will see that such reasoning would later lead Dijkstra to write a correctness proof of an incorrect algorithm.

Although not asserted by Dijkstra, Dekker's algorithm is starvation free as well as deadlock free. In more than one place, Dijkstra mentioned the difficulty of finding and proving the correctness of concurrent algorithms, and he urged the reader to try to solve a problem before reading his solution. In that spirit, I leave the proof that Dekker's algorithm is starvation free to the reader.

4.2.3 Dijkstra's Algorithm

The *N* process mutual exclusion algorithm of Dijkstra [1965] uses Dekker's idea of a variable *turn* that determines which among competing processes should be the next to enter the critical section. Instead of having the value of *turn* set by a process upon exiting the critical section, a process trying to enter the critical section tries to set *turn* to its own number. It can do so only when *turn* is the number of a process currently in its noncritical section. For this purpose, the algorithm uses an additional variable *idle*, where *idle*[*i*] equals **true** when process *i* is in its noncritical section. The code is in Figure 4.3. The variable *temp* is declared to be local to the process, each process having its own copy. The expression $(1..N) \setminus \{i\}$ equals the set of all integers from 1 through *N* except *i*.

The algorithm obeys the one-bit protocol, since a process enters the critical section only by completing the **else** clause without looping back to *L*. Mutual exclusion is therefore guaranteed. To show deadlock freedom, it suffices to assume some process is trying to enter the critical section and no process ever succeeds, and to show that some process eventually does enter the critical section. With no process entering the critical section, eventually every process either remains forever in its critical section or keeps looping through the **if** test at *L*. Since *idle*[*j*] is **true** if process *j* is in its noncritical section, if process number *turn* is not looping, then some looping process will set *turn* to its own number. At that point, *turn* must always equal the number of a looping process, so *idle*[*turn*] always equals **false**. Each looping process can then set *turn* at most once, so eventually the value of *turn* stops changing. The value of *flag*[*j*] will then eventually forever equal **true** for every process except process *turn*, and process *turn* will enter its critical section.

```
variables $flag[i \in 1..N] = \textbf{true}$, $idle[i \in 1..N] = \textbf{true}$,  $turn \in 1..N$
process $i \in 1..N$
variable $temp \in 1..N$
  do while true
       do     noncritical section ;
              $\langle idle[i] := \textbf{false} \rangle$ ;
          L:  if $\langle turn \neq i \rangle$
                then $\langle flag[i] := \textbf{true} \rangle$ ;
                     $\langle temp := turn \rangle$ ;
                     if $\langle idle[temp] \rangle$ then $\langle turn := i \rangle$ fi ;
                     goto $L$
                else $\langle flag[i] := \textbf{false} \rangle$ ;
                     for $j \in (1..N) \setminus \{i\}$
                       do if $\langle \neg flag[j] \rangle$ then goto $L$ fi od ;
              fi ;
              critical section ;
              $\langle flag[i] := \textbf{true} \rangle$ ;
              $\langle idle[i] := \textbf{true} \rangle$
       od
  od
```

Figure 4.3 Dijkstra's algorithm.

While it is deadlock free, Dijkstra's algorithm is not starvation free even for two processes. One process *i* can set *turn* equal to *i* and keep cycling through its noncritical and critical sections, while another process waits forever, always reading *idle*[*i*] equal to **false**.

4.2.4 Semaphores

These mutual exclusion algorithms were of intellectual interest only. They were impractical for implementing mutual exclusion in the THE system. Instead, Dijkstra introduced the semaphore, described in EWD35. A binary semaphore *sem* is a special kind of variable, whose value initially equals 0, that can be accessed only by the two operations *P*(*sem*) and *V*(*sem*).[5] These operations are defined by:

$$P(sem): \left\langle \begin{array}{l} \textbf{await } sem > 0\,; \\ sem := sem - 1 \end{array} \right\rangle$$

$$V(sem): \langle\ sem := 1\ \rangle$$

Thus, the value of a binary semaphore always equals either 0 or 1. As reported in EWD123, Carel Scholten suggested allowing semaphores whose value can be any natural number, where the *V*(*sem*) operation is replaced by $\langle\, sem := sem + 1\, \rangle$.

[5] *P* and *V* are the first letters of Dutch words meaning *pass* and *release*.

It is trivial to implement mutual exclusion with a binary semaphore *sem*. Simply precede the critical section with a *P*(*sem*) statement and follow it with *V*(*sem*). Semaphores can also be used in other ways to synchronize processes. Binary semaphores are still commonly used today. They are now called locks, and *P* and *V* are called *lock* and *unlock*.

When multiple processes are waiting to execute a *P*(*sem*) operation, execution of a *V*(*sem*) allows one of them to proceed. Dijkstra said nothing about which one will enter. His definition allows an individual process to wait forever if other processes keep executing *P*(*sem*) operations. The simple mutual exclusion algorithm using a single semaphore is then deadlock free but not starvation free.

Allowing a process to wait forever on a *P*(*sem*) operation may not be acceptable. A semaphore *sem* is said to be *fair* if every process waiting on a *P*(*sem*) operation will eventually execute it, assuming every *P*(*sem*) operation is followed by a corresponding *V*(*sem*) operation. Fair semaphore implementations often guarantee something stronger than fairness, ensuring that no process waits too long to execute a *P*(*sem*) operation. It would be surprising if that were not the case for semaphores in the THE system. For proving properties such as starvation freedom that assert something eventually happens, fairness is usually all that is required.

4.2.5 A Closer Look at the Problem

We have seen Dijkstra's solution of the mutual exclusion property. More important than his solution was the statement of the problem, including what could be assumed about the processes. He first stated the mutual exclusion requirement. He then assumed that processes communicated through what are now called atomic shared memory registers, accessed only with read and write operations. (Remember that he wrote an **await** statement as a waiting loop.) During an execution of the algorithm, the results of operations to shared memory by all the processes were assumed to be the same as if those operations were performed in some sequential order consistent with each process's code. He then stated four requirements for a solution. The last three were:

(1) Nothing could be assumed about the relative execution speeds of the processes. Left implicit was the assumption that the speed was not zero—that is, each process kept executing instructions unless it executed a **halt** instruction.

(2) Any process could halt in its noncritical section.

(3) Deadlock freedom. In stating this assumption, he explicitly ruled out any algorithm in which *"After you. No, after you."* blocking, though highly improbable, could continue indefinitely.

The importance of each of these requirements is remarkable. Requirement 1 was probably suggested by the THE system, in which all processes shared a single processor, so one process might do nothing for a long time while other processes used the processor. However, Dijkstra realized this was not just an artifact of that system. He understood that the distinction between process and processor would exist even on multiprocessor machines, and that processes could represent devices with very different execution speeds, such as computers, printers, and humans. The assumption that each process keeps executing, today called *process fairness*, is now assumed for most concurrent algorithms.

Requirement 2 ruled out simple solutions in which all processes cycle through the critical section—a process immediately exiting it if it has nothing critical to do.

The remarkable part of requirement 3 is not deadlock freedom but that it should hold for every possible execution. Using the 1-bit protocol, it's not hard to find algorithms that ensure mutual exclusion but allow the kind of blocking the requirement forbids. Dijkstra probably thought some people would think those algorithms were correct because such blocking is unlikely to persist for long.

We now take it for granted that correctness means correctness of all executions. But this paper was published two years before Floyd's [1967] seminal paper that effectively launched the modern field of program correctness. I don't know of any algorithm that was published before this one with a correctness proof. Even 14 years later, no indication of why an algorithm might be correct was required for the ACM to publish it [Krogh 1979]. Testing was the standard way of verifying correctness of an algorithm. But testing cannot show that every possible execution of an algorithm does what it is supposed to—especially a concurrent algorithm, where the absence of a bound on relative execution speeds of the processes usually allows infinitely many possible executions. In 1965, only a proof could do that.

Dijkstra's remaining requirement, the first one on his list, was:

> The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

I used to think that this was added as a "poor man's substitute" for starvation freedom. It meant that, although his solution didn't guarantee that any particular process ever entered the critical section, it ensured that every process had an equal chance of entering it. But in EWD35, written three years earlier, Dijkstra mentioned that Dekker's algorithm was symmetric but did not say that it was starvation free. He apparently felt this condition was more important than starvation freedom. He could have known the one-bit algorithm and not mentioned it because it was asymmetric.

The first starvation free N-process mutual exclusion algorithm, which was also symmetric, was published eight months later by Knuth [1966]. It essentially uses Dekker's idea of modifying the one-bit algorithm so process priority is indicated by a variable *turn* that is set by a process when exiting the critical section. The process numbers, in order of decreasing priority, are $turn, turn - 1, \ldots, 1, N, N - 1, \ldots, turn + 1$.

Almost all mutual exclusion algorithms published since then have been starvation free. Symmetry no longer seems to be considered important. For example, the popular bakery algorithm [Lamport 1974a] ensures first-come, first-served entry to the critical section—arguably a fairer guarantee than symmetry—but is not symmetric because of a static priority among processes that arrive concurrently.

4.2.6 The Dining Philosophers

The mutual exclusion problem has been generalized in various ways to allow more than one process to be in its critical section at the same time. One early example is when the critical section serves to protect some data. A process must not modify the data while another process is trying to read or modify it. However, multiple processes that only read the data can access it at the same time. This is called the *readers/writers* problem. Processes are partitioned into readers and writers, and the mutual exclusion criterion is that a writer cannot be in its critical section while any other process is in its critical section. Early solutions used semaphores [Courtois et al. 1971].

The most popular variant of mutual exclusion is the Dining Philosophers problem. It appears in EWD198, where Dijkstra calls it the "Problem of the Dining Quintuple". EWD1000 recounts that he invented it as an examination problem for a course he gave in 1965, and that C.A.R. (Tony) Hoare renamed it the Dining Philosophers problem.

In this problem, five philosophers alternate between thinking and eating. The philosophers eat at a round table, where each is assigned a seat with a plate in front of him. They are served a "difficult kind of spaghetti" that must be eaten with two forks.[6] There are five forks, one between each pair of philosophers. A philosopher can use only the two forks next to him; and he eats when in his critical section. Since a fork can be used by only one philosopher at a time, the mutual exclusion condition is that two adjacent philosophers cannot be in their critical sections at the same time.

[6] The insular nature of European cuisine at the time is displayed by the philosophers' use of two forks rather than a pair of chopsticks.

Dijkstra evidently regarded this as a concurrent programming problem rather than an algorithm problem. In Dijkstra [1971], he derived a solution using a global semaphore that allows only one philosopher at a time to examine the state of his two neighbors and modify his own state. It was an exercise in practical programming, not in finding an interesting algorithm. Dijkstra then observed that his solution was probably not satisfactory because a philosopher could be prevented from ever eating by voracious philosophers on either side of him. He said that a solution should allow every hungry philosopher to eventually enter his critical eating section, making it "starvation free". He indicated how such a solution can be obtained but did not describe it.

The dining philosophers became very popular, both as a programming problem and an algorithmic one. I never felt it merited as much attention as it received; I believed its popularity was due to the cute story with which it was presented. A number of years later, my colleagues and I did some work that I thought deserved to be well known. The example of the dining philosophers inspired me to explain it with a story involving traitorous generals [Lamport et al. 1982]. That worked quite well.

4.3 Self-stabilization

4.3.1 The Problem

Nine years passed before Dijkstra published his second paper about concurrent algorithms, which also appeared in CACM [Dijkstra 1974]. I believe it was the first paper devoted to fault tolerance in concurrent algorithms.

Dijkstra observed that correct functioning of a system can be achieved by ensuring that it is always in a "legitimate" state. A fault in the program or the computer can cause incorrect behavior by putting the system in an illegitimate state. The system could recover from the fault by correcting its state.

An obvious way for the system to correct its state is to periodically check it and reset an illegitimate state to a legitimate one. This is not easy in a multiprocess system, where each process can examine only part of the state. Dijkstra's idea was to have each process, acting independently, cause the system to eventually go from any state to a legitimate one.

He defined a *self-stabilizing* algorithm to be one that, when started in any possible state, would eventually reach a legitimate state and operate properly from then on. He decided to take as his goal to make self-stabilizing what we would now call a token ring.[7] As formulated by Dijkstra, there are $N + 1$ processes numbered

[7] The paper states that one of its algorithms was used shortly after being discovered, suggesting that Dijkstra chose a token-ring algorithm because he had an application in mind.

0 through N, arranged in a circle. Let $L(i)$ and $R(i)$ equal $(i-1)\bmod(N+1)$ and $(i+1) \bmod (N+1)$, respectively. We call processes $L(i)$ and $R(i)$ the left and right neighbor of process i. In a token ring, there should always be a single token that resides at one of the processes. When process number i has the token, it must pass it to its right-hand neighbor, process number $R(i)$.

Expressed more precisely, the problem is to find an algorithm and a Boolean-valued function $HasToken(i)$ of the state of process i such that process i can perform an action only when $HasToken(i)$ equals **true**. A legitimate state is one in which $HasToken(i)$ equals **true** for exactly one process i, in which case its action makes $HasToken(i)$ equal **false** and $HasToken(R(i))$ equal **true**. Self-stabilization means that started in any possible state, the algorithm must eventually reach a legitimate state.

4.3.2 A Coarse-grained Algorithm

Dijkstra devised three different self-stabilizing token ring algorithms. I will discuss only the first. It has a single array variable S indexed by process number, where $S[i]$ is in $0..(K-1)$ for a positive integer K whose value is constrained below. A legitimate state is one in which there is some process number i such that, for every process number j:

$$S[j] = \begin{cases} S[0] & \text{if } 0 \leq j \leq i \\ (S[0]+1) \bmod K & \text{if } i < j \leq N \end{cases}.$$

For i equal to N, this means all the $S[j]$ are equal.

Dijkstra defined $HasToken$ by:

$$HasToken(i) = \begin{cases} S[i] \neq S[L(i)] & \text{if } i \neq 0 \\ S[i] = S[L(i)] & \text{if } i = 0 \end{cases}.$$

Remember that $L(0)$ equals N, so $HasToken(0)$ equals $S[0] = S[N]$.

It is easy to check that, in any legitimate state, $HasToken(i)$ equals **true** for exactly one process i. It is also easy to check that a process i other than 0 can pass the token to process $R(i)$ only by setting $S[i]$ equal to $S[i-1]$. Process 0 can pass the token to process 1 by changing $S[0]$ to $(S[0]+1) \bmod K$. Each token-passing action starting in a legitimate state then produces a legitimate state.

Dijkstra begins with a coarse-grained algorithm in which the entire operation of passing the token from i to $R(i)$ is a single action. We've just seen that this algorithm correctly implements a token ring when started in a legitimate state. We now need to verify self-stabilization. To do this, for each process i, we allow the initial

```
variables $flag[i \in 1..N] =$ true, $idle[i \in 1..N] =$ true,  $turn \in 1..N$
process $i \in 1..N$
variable $temp \in 1..N$
  do while true
     do    noncritical section ;
           $\langle idle[i] :=$ false$\rangle$ ;
        L: if $\langle turn \neq i \rangle$
             then $\langle flag[i] :=$ true$\rangle$ ;
                  $\langle temp := turn \rangle$ ;
                  if $\langle idle[temp] \rangle$ then $\langle turn := i \rangle$ fi ;
                  goto $L$
             else $\langle flag[i] :=$ false$\rangle$ ;
                  for $j \in (1..N) \setminus \{i\}$
                    do if $\langle \neg flag[j] \rangle$ then goto $L$ fi od ;
           fi ;
           critical section ;
           $\langle flag[i] :=$ true$\rangle$ ;
           $\langle idle[i] :=$ true$\rangle$
     od
  od
```

Figure 4.4 The coarse-grained algorithm.

value of $S[i]$ to be any number in $0..(K-1)$ and we show that the algorithm eventually reaches a legitimate state. The precise algorithm is in Figure 4.4. The algorithm assumes $K \geq N$. Here is Dijkstra's proof of self-stabilization.

(1) In any possible state, *HasToken*(i) is true for at least one process i.

Proof. If *HasToken(i)* is false for each $i > 0$, then $S[i-1] = S[i]$ for all $i > 0$. This implies $S[0] = S[N]$, which implies *HasToken(0)* is true because $L(0) = N$. ■

(2) After an action of process i has occurred, another action of process i cannot occur until the value of $S[L(i)]$ has changed.

Proof. Executing the process i action makes *HasToken*(i) false. It can become true again only by $S[L(i)]$ changing. ■

(3) Infinitely many actions of every process occur.

Proof. By step 1, it is always possible for an action of some process to occur, so infinitely many actions of some process must occur. Since the value of $S[j]$ is changed only by an action of process j, step 2 implies that if infinitely many actions of any process i occur, then infinitely many actions of $L(i)$ occur. Hence, infinitely many actions of any one process implies infinitely many actions of all processes. ■

(4) Eventually, a process 0 action that sets $S[0]$ to 0 occurs.

Proof. By step 3, infinitely many process 0 actions occur, and each such action increments $S[0]$ by 1 modulo K. ■

Define the color of each process according to the following rules:

- Initially all processes are white.
- The first process 0 action that sets $S[0]$ to 0 turns process 0 blue. (Such an action occurs by step 4.)
- Node $i > 0$ is set blue either if process i executes an action when node $L(i)$ is blue, or if $L(i)$ is set blue and $S[L(i)]$ then equals $S[i]$. (Thus a process action that sets itself blue may also set one or more processes to its right blue.)

(5) After process 0 becomes blue, at most $N - 1$ process N actions can occur before the action that turns N blue.

Proof. Because values of $S[i]$ travel from left to right between processes 0 and N, and a value moving from a blue node to a white node turns the white node blue, step 2 implies that the values that $S[N]$ can have before N becomes blue must be among the white nodes to its left. When process 0 first becomes blue, there are at most $N - 1$ white nodes to the left of node N. Hence, at most $N - 1$ process N actions can occur before N becomes blue. ■

(6) Immediately after N becomes blue, the sequence $S[0], S[1], \ldots, S[N]$ is nonincreasing.

Proof. Values move left to right, and new values are created by process 0 actions that increment $S[0]$ by 1 modulo K. By 2 and 5, until N becomes blue, at most $N - 1$ process N actions have occurred. Since $S[0]$ equals 0 when process 0 becomes blue, we can use the assumption $K \geq N$ to deduce from this that the value of $S[0]$ can be at most $K - 1$. Therefore, up to and immediately after execution of the action that makes N blue, the sequence of values of blue nodes is nonincreasing. ■

(7) The algorithm is eventually in a legitimate state.

Proof. By step 6, eventually $S[0], S[1], \ldots, S[N]$ is nonincreasing. It is easy to see that this must remain true until a process 0 step decreases $S[0]$, which by step 3 must happen and by definition of the process 0 action can happen only after $S[N]$ equals $S[0]$. But $S[N] = S[0]$ and the sequence of values $S[i]$ being nondecreasing implies that all the $S[i]$ are equal, which is a legitimate state. ■

4.3.3 A Finer-grained Algorithm

Dijkstra next considered the finer-grained algorithm obtained by modifying the algorithm in Figure 4.4 to make the **await** statement and the **if** statement separate atomic actions—in other words, adding "$\rangle$" before the ";" and "$\langle$" before the "**if**". That modification to the code in Figure 4.4 would be adequate to describe the algorithm starting from a legitimate state. However, self-stabilization also requires showing that a legitimate state is reached from any possible starting state. The state of the finer-grained algorithm consists not only of the value of the array S but also the control state of each process—that is, whether the next action of a process is execution of the *await* statement or the **if** statement. The algorithm should be self-stabilizing if the algorithm is started with each process in either possible control state.

Writing such an algorithm in pseudo-code would be complicated without additional language features. (It is easy to do with more mathematical ways of describing algorithms.) So, we will make do with this informal description of the finer-grained algorithm.

The proof of self-stabilization is almost identical to the proof for the coarse-grained algorithm. That proof breaks down in step 7. The assertion that a process 0 step can change $S[0]$ only when $S[0]$ equals $S[N]$ is incorrect. Suppose K equals N. It is possible that $S[N]$ equaled $N-1$ immediately before process N became blue. At that time, $S[0]$ could have equaled $N-1$ and found its **await** condition to be true, moving control to its **if** statement. Process N could then have become blue with $S[N]$ being set to 0. Process 0 could then have set $S[N]$ to $(N-1+1)$ mod N (since K equals N), which equals 0, and $S[0], S[1], \ldots, S[N]$ could then be nonincreasing, invalidating the proof.

This problem can be corrected by assuming $K > N$, and indeed this assumption implies that the algorithm is self-stabilizing. However, I hope that my admonition in Section 4.2.2 about the unreliability of behavioral reasoning has made the reader skeptical of my assertion that this correction produces a correct proof. There are now more reliable ways of proving properties like self-stabilization. However, they are beyond the scope of this chapter. The proofs presented here reflect the ones Dijkstra wrote, which today are best considered informal explanations rather than rigorous proofs.

4.3.4 What Dijkstra Actually Wrote

Dijkstra first discussed self-stabilization in EWD391, where he presented the coarse- and finer-grained algorithms and their proofs discussed above. However, he did not describe those two versions of the algorithm in terms of atomic actions but in terms of "demons" that control what action gets performed next. His first algorithm was obtained by a single centralized demon, and the second by what he called a “distributed” demon.

It is easy to see that the centralized demon produces the algorithm of Figure 4.4. However, there seem to be two reasonable interpretations of what execution by a distributed demon could mean. The first is the finer-grained algorithm discussed above. The second is to consider that the token-advancing operation for process $i \neq 0$ is performed by reading $S[L(i)]$ just once, rather than once for evaluating the **await** and again for performing the assignment to $S[i]$. In that case, the finer-grained version should have had an additional variable local to each process that stores the value of $S[L(i)]$ read in the **await** statement for use in the **if** statement.

Fortunately, there is now an easy way to determine which one Dijkstra meant: using a model checker to verify correctness of the algorithms for small values of N and K. A model checker checks the correctness of all possible executions, usually of a small instance of an algorithm. Model checking shows that the version that reads $S[L(i)]$ just once is not self-stabilizing for $K = N + 1 = 4$. I conjecture that it is self-stabilizing for large enough values of K, but I will leave the proof or disproof of this as an exercise for highly motivated readers.

The description in terms of demons was not just ambiguous. It seems to have confused Dijkstra himself. EWD391 contained a proof of the finer-grained algorithm based on arguing that executing the algorithm with the distributed demon was equivalent to executing it with the centralized demon. That argument led to a “proof” that the finer-grained algorithm is self-stabilizing if $K \geq N$. EWD392 contains an erratum stating that the algorithm with a distributed demon requires the assumption $K > N$, but says nothing about requiring any change to the proof. There is a note crediting Carel Scholten with pointing out the error, saying “[It] shows a serious flaw in my reasoning; I should have known better!”

The other two algorithms in the CACM paper and their proofs first appeared in EWD392 and EWD396. However, no proofs were included in the published version, which was only $1\frac{3}{4}$ pages long. The word “proof” does not even appear.[8] Dijkstra certainly recognized the need for correctness proofs of concurrent algorithms. Perhaps he omitted the proof because he felt that a shorter paper would have more impact.

4.3.5 The Paper’s Influence

The coarse-grained algorithm described above requires at least N states per process. The coarse-grained version of the second algorithm in the CACM paper

[8] The submitted version contained a comment about Scholten having found a nice proof for one of the algorithms, but the comment was eliminated in the published version.

requires only 4 states per process, and that of the third algorithm requires only 3 states. (The finer-grained versions of these algorithms require $N+1$, 4, and 3 states, plus the extra control state that is not accessed by other processes.)

I regard these algorithms to be the most brilliant work Dijkstra ever did. I was amazed by their simplicity and their depth. I was also impressed by the significance of the CACM paper. It provided the first rigorous approach to fault-tolerance in concurrent systems—indeed, the first discussion of the problem.

Unfortunately, very few others understood the paper's significance. This was probably due to how little motivation the paper provided for self-stabilization. The original submission (EWD397) said only that "the appreciation is left as an exercise to the reader." Probably in response to referees' comments, Dijkstra added two introductory paragraphs, but they mostly clarified the problem and provided just a bit of motivation. As a result, that paper almost completely disappeared until 1983.

In 1983, I gave an invited talk at the *Principles of Distributed Computing* (PODC) conference. In it, I talked briefly about the CACM paper and said how brilliant and how important it was. This introduced self-stabilization to the PODC community and led to it becoming an active research topic. I am proud of the part I played in helping this paper receive the attention it deserved. A discussion of the work the paper spawned as well as an explanation of the 4-state algorithm are in Chapter 5 written by Ted Herman.

The CACM paper also had an important effect on me personally. I discovered a generalization of the 4-state algorithm to a tree of processes [Lamport 1974b] and sent it to Dijkstra. This led to his sending me some of his EWDs, which led to my involvement in the paper discussed next.

4.4 On-the-fly Garbage Collection

It was probably in April of 1975 that I received from Dijkstra a copy of EWD492, *On-the fly garbage collection: an exercise in cooperation*. I wrote to him suggesting a small simplification, and about a month later I received a copy of EWD496 with the same title, listing me among its five authors. A much-revised version was published in CACM in November 1978 [Dijkstra et al. 1978]. The story of how EWD496 became the CACM paper is instructive.

4.4.1 The Problem

In 1975, garbage collection was relevant mainly for LISP programs. Today, it arises in managing a heap of objects, which is at the heart of implementing object-oriented programs. The value of a program variable can be a pointer to an object in the heap, and a heap object can contain pointers to other heap objects. A heap object is said to be *reachable* if and only if it is pointed to by a variable or a reachable heap object. Only reachable objects can ever be used by the program; non-reachable objects are said to be *garbage*. Garbage collection is the reclaiming of memory space occupied by garbage objects.

Traditionally, garbage collection has been done by pausing the program, collecting the garbage, and then restarting the program. An interactive program becomes unresponsive while the garbage is being collected. The paper addresses the problem of eliminating this interruption by having garbage collection performed continuously by a *collector* process that runs concurrently with the main program. To eliminate details that do not affect the basic concurrency problem, the paper makes the following simplifying assumptions:

- There is a fixed set of variables that can contain pointers to heap objects.
- The size of the heap is fixed.
- All objects occupy the same amount of memory and have the same number of pointers.

The first assumption allows us to replace pointer-containing variables with special *root* objects that are, by definition, always reachable. The other assumptions mean there is no need to consider object creation and destruction. We can assume a fixed set of objects. The main program obtains new objects from a portion of the heap called the *free list*, reachable from special roots. Obtaining a new object then becomes an ordinary operation of the main program on reachable heap objects. Garbage collection consists of making garbage objects reachable as part of the free list. Real programs can contain null pointers, which don't point to any object. They can be considered pointers to a special root object called *null*.

The result of these simplifications is that all operations performed by the main program that change the heap can be represented as instances of one simple operation: setting a pointer of a reachable object to point to a reachable object.

I now switch to notation similar to that used in the paper. The state of the heap is described as a directed graph with nodes and edges. If a pointer in object *A* points to object *B*, we say that there is an edge from node *A* to node *B*. A node is reachable if and only if it is reachable from a root node, under the usual definition of reachability in a directed graph.

Let *Nodes* be the (fixed) set of all nodes. The heap is represented by a variable *heap* whose value is an array indexed by the set *Nodes*. We are assuming that all nodes have the same number of outgoing edges, so we can represent the outgoing edges from a node *A* as an array *heap*[*A*].*edges* of nodes indexed by a fixed set *EdgeIds* of edge identifiers. We represent the main program by a process called the *mutator*, and we call the garbage collecting process the *collector*. The mutator executes a sequence of operations

$$heap[A].edges[e] \; := \; B$$

for some reachable nodes A and B and some e in *EdgeIds*. The collector executes a sequence of *collections*, each of which can add unreachable nodes to the free list, making them reachable.

To represent the programming operation of creating a new object and making it reachable by mutator operations, the collector and mutator must obey some protocol for adding nodes to and removing them from the free list. The paper points out that this can be viewed as an instance of producer/consumer synchronization, which had known solutions [Lamport 1977]. We ignore how the free list is implemented by simply assuming that the collector can add a node to the free list with an atomic action, making it reachable with all its edges pointing to already reachable nodes. A collection will consist of a *marking* phase in which reachable nodes are identified, followed by a *freeing* phase in which nodes found not to be reachable are put on the free list.

There are two correctness conditions required of a solution:

(CC1) Every garbage node is eventually added to the free list.
(CC2) The collector makes no change to the heap except by putting garbage nodes on the free list.

Condition CC1 should be strengthened to require that garbage is collected in a timely fashion. The paper's algorithm ensures that any node that is garbage at the beginning of the freeing phase will be added to the free list by the end of the following complete collection. I will not discuss the proof of CC1 and will consider only CC2. Since there is no reason for the collector to modify the heap other than by adding nodes to the free list, the only nontrivial part of satisfying CC2 is ensuring that no reachable node is ever added to the free list.

4.4.2 A Solution

Dijkstra wanted to minimize any cost to the mutator required by the algorithm. To avoid synchronization costs, he wanted the only atomic operations to the heap to be reading or changing a single edge of a single node. With that restriction, it's easy to construct scenarios in which a node remains reachable even though the collector never sees any edge pointing to it. So, in addition to changing edges, the mutator must inform the collector in some way when it changes them. This is done by having a flag, $heap[A].white$, initially equal to **true**, for every node A. (The flag is not considered to be part of the heap.) In the solution originally submitted

```
   for $R \in Roots$ do $\langle heap[R].white := \mathbf{false} \rangle$ od;
L: for $A \in Nodes$
     do if $\neg(heap[A].white \lor heap[A].black)$
          then for $e \in EdgeIds$ do $\langle temp := heap[A].edges[e] \rangle$ ;
                                       $\langle heap[temp].white := \mathbf{false} \rangle$
                                    od ;
               $heap[A].black := \mathbf{true}$ ;
               goto $L$
        fi
     od
```

Figure 4.5 The collector's marking phase.

to CACM, the mutator's single operation consists of these two atomic actions, for some reachable nodes A and B and some edge identifier e:

$$\begin{aligned} &\langle heap[B].white \ := \ \mathbf{false}\rangle; \\ &\langle heap[A].edges[e] \ := \ B\rangle. \end{aligned}$$

Thus, the only added cost for the mutator is setting one bit during every pointer-changing operation. There is also a *black* flag for each node, used only by the collector. We call a node A black if *heap*[A].*black* equals **true**. A non-black node is said to be white if *heap*[A].*white* equals **true** and to be gray if it equals **false**.[9] Initially, no node is black; and the collector's algorithm ensures that a node can be black only when the collector is performing a collection.

The collector's marking phase is shown in Figure 4.5, where *Roots* is the set of root nodes and *temp* is a variable local to the collector. The phase begins with the collector setting the roots gray. In the body of the **for** A loop, if node A is gray, then the collector: (i) sets the *white* flag false for every node pointed to by A, making that node gray if it was white, otherwise leaving its color the same; (ii) makes node A black; and (iii) begins the **for** A loop again by going back to control point L. Thus, the marking phase ends when the collector examines all nodes without finding a gray one. This must eventually happen because each execution of that loop makes one gray node black.

To prove CC1, we must show that no reachable node is ever put on the free list. This requires showing that, at the end of the marking phase, every white node is garbage. Here is a sketch of Dijkstra's proof.

[9] The paper describes the algorithm in terms of these three colors, without specifying how they are encoded. I prefer using the two bits because it makes clear that setting a node black or non-black modifies data accessed only by the collector, and that a node is made non-white by setting a bit without having to read it.

(1) While the collector is in the marking phase, no black node ever points to a white node.

Proof. When a node A is turned black, the *white* flag of all the nodes pointed to by its edges had just been set false. If the mutator changed one of those edges before the collector made A black, it must have first set to false the *white* flag of the node the edge currently points to. ■

(2) While the collector is in the marking phase's **for** A loop, every reachable node is reachable by a path from a black or gray node.

Proof. Every reachable node is reachable from a root node, and every root node has been turned grey and must remain either gray or black. ■

(3) Upon completion of the marking phase, there is no gray node.

Proof. Suppose there is a gray node upon completion. It must have been white and become gray after being examined in the **for** A loop. Consider when the first such node became gray. By steps 1 and 2, it must then have been pointed to by a gray node that had not yet been examined by the loop. This is impossible because, when the loop examined that gray node, the **for** A loop would have been restarted. ■

(4) Upon completion of the marking phase, no white node is reachable.

Proof. If there were a white reachable node at that point, consider the one reachable by the shortest path from a root. By steps 2 and 3, it is reachable by a path from a black node, contradicting step 1. ■

The proof, sketched here and presented in more detail in the paper submitted to CACM, convinced Dijkstra and his four coauthors. It is wrong, and the algorithm is incorrect. Here is a scenario that displays the error.

The mutator begins the operation of making an edge from node A point to node B, making B gray. The collector then performs a complete collection, leaving the reachable nodes A and B both white. It then starts another collection, reaching a point in which it has made A black but has not yet examined B or made it non-white. The mutator then makes the edge of A point to B. There is now an edge from a black to a white node, showing that step 1 of the alleged proof is false. I will leave it to the reader to complete the scenario to one in which the collector finishes the marking phase, leaving B white, and then adds it to the free list in the freeing phase.

4.4.3 Verification

That five computer scientists, including someone as careful as Dijkstra, were fooled by an incorrect proof indicated that we needed more reliable methods of reasoning about concurrent algorithms. Edward Ashcroft [1975] had recently developed one method.

The simplest semantic view of concurrent algorithms that formalizes how Dijkstra reasoned about them is that an execution of an algorithm is a sequence of states, and a correctness condition is a predicate on such sequences. The type of correctness property at the heart of virtually all rigorous correctness proofs is invariance, which asserts that a predicate on states is true for every state of every execution. For example, step 1 of the incorrect proof asserts that this predicate on states is invariant:

MP. If the collector is in a marking phase, then no edge from a black node points to a white node.

Ashcroft observed that we can prove that a state predicate P is invariant by verifying these two conditions:

(1) P is true of every initial state.
(2) Any atomic action of the program beginning in any state satisfying P produces a state that satisfies P.

That these conditions imply P is true of every state of every execution follows by mathematical induction on the number actions required to reach the state.

A predicate P satisfying these two conditions is called an inductive invariant. Not every invariant is inductive. For example, even if *MP* were an invariant, it could not be inductive because condition 2 is not true for a state with a black node A and a white node B in which the mutator is about to perform the action that makes an edge of A point to B. To prove that a formula is invariant, we must usually find an inductive invariant that implies it. To prove the invariance of *MP*, the inductive invariant would have to imply that, if the mutator is in a state in which it could make an edge of a black node A point to node B, then B is not white. An attempt to find such an inductive invariant would have failed and would have led to the counterexample. This actually happened after the fact.

I learned that there was an error in the algorithm by receiving a copy of a letter Dijkstra had sent to the editor of CACM withdrawing the paper. The letter said that M. Woodger had found the error but gave no indication of what the error was. Since Dijkstra's proof was so convincing, I thought that the error must be minor and easily corrected. So, I decided to write an inductive invariance proof, thinking that I would then find and correct the error. In about 15 minutes, trying to write the proof led me to the error.

I suspected that the algorithm could be fixed by changing the order in which two atomic actions were performed. I had no good reason to believe that would work, and I could see no informal argument to show that it did. However, I decided to go ahead and try to find and prove the correctness of a suitable inductive invariant implying that only garbage nodes are white in the freeing phase. It took me about two days of solid work, but I did it. I was then convinced that the resulting algorithm was correct, but I still had no intuitive understanding of why it was correct.

Meanwhile, Dijkstra had found a different fix and had written the same kind of proof as before. There was no reason not to use his algorithm, and I sketched an inductive invariance proof of it. Given the evidence of the unreliability of his style of proof, I tried to get Dijkstra to put a rigorous inductive invariance proof in the paper. He was unwilling to do that, though he did agree to make his proof closer to an invariance proof, with less behavioral reasoning. Here are his comments on that, written in July 2000 in a letter to me:

> There were, of course, two issues at hand: (A) a witness showing that the problem of on-the-fly garbage collection with fine-grained interleaving could be solved, and (B) how to reason effectively about such artifacts. I am also certain that at the time all of us were aware of the distinction between the two issues. I remember very well my excitement when we convinced ourselves that it could be done at all; emotionally it was very similar to my first solutions to the problem of self-stabilization. Those I published without proofs! It was probably a period in my life that issue (A) in general was still very much in the foreground of my mind: showing solutions to problems whose solvability was not obvious at all. It was more or less my style. I had done it with the mutual exclusion [algorithm].

Following Ashcroft, a few methods were proposed for verifying the two conditions needed to prove inductive invariance, their differences being based on the way the algorithm was described. I used one I had just devised [Lamport 1977] in my proof. The one developed by Susan Owicki and David Gries [Owicki and Gries 1976] was the most popular method, probably because they described the algorithm with ordinary programming-language constructs. Gries later used it to prove the published algorithm [Gries 1977]. His proof was essentially the same as the one I had sketched. He simplified things a bit by combining two atomic operations into one,[10] but it would have been easy to add the details needed to handle the actual algorithm.

4.4.4 The Algorithm and its Significance

The correct algorithm of the published paper is obtained from the incorrect one by a simple but counterintuitive change to the mutator. The first atomic action of

[10] He mentioned this simplification in a footnote that CACM failed to print, though it did include the footnote number in the text.

the mutator's operation of making an edge of node A point to node B is changed so that, instead of setting to **false** the *white* flag of node B, it sets to **false** the *white* flag of the previous node to which the mutator made an edge point.

The paper provides a path of reasoning that leads to the algorithm and a correctness proof. It calls this proof an "informal justification which we do *not* regard as an adequate substitute for a formal correctness proof." In the text of a talk Dijkstra gave in 1984, he wrote this about concurrent and distributed systems:

> I know of only one satisfactory way of reasoning about such systems: to prove that none of the atomic actions falsifies a special predicate, the so-called 'global invariant'. Once initialized, the global invariant will then be maintained by any interleaving of the actions. [Dijkstra 1984]

This is the only place I know in which he explicitly discussed the concept of an inductive invariant.

Dijkstra was wise to decide that the important contribution of the paper was the algorithm, not its proof of correctness. The algorithm implements the sharing of data, the heap, by concurrently executed processes using only atomic read and write operations. It does not use mutual exclusion or any synchronization primitive such as a semaphore. This means that neither process ever has to wait for the other one (except if the mutator finds the free list empty, when waiting is unavoidable). Such an algorithm is now called *wait-free* [Herlihy 1991]. This was the first nontrivial[11] wait-free implementation of a shared data structure ever published.

4.5 Termination Detection

The last two of the five concurrent algorithm papers to be examined each had an algorithm for detecting termination in distributed computations. As documented in Section 4.4.3, Dijkstra was interested in finding concurrent algorithms to solve problems for which it was not clear that a solution existed. I think it must have been clear to him that solutions to this termination detection problem existed. He was probably interested less in the algorithms than in deriving them. He presented the algorithms with derivations based on inventing the invariants that the algorithms should maintain. I suspect he "derived" the algorithms after inventing them. But our concern here is the algorithms, not how they might be derived. The algorithms are therefore presented without rigorous derivations.

[11] I apparently invented an earlier wait-free implementation of a bounded FIFO queue for use as an example in Lamport [1977], but it seemed so trivial that I assumed it was already known.

4.5.1 The Problem

Assume a set of processes that collectively perform a terminating computation, communicating with one another by sending messages. The content of the messages and what is being computed are irrelevant. All that we care about is that a process can be in one of two states: *idle* or *active*. An active process can send messages to other processes. A process can change from idle to active only when it receives a message. An active process can become idle at any time. The computation terminates when all processes are idle.

One process is designated the *leader*. The problem is to superimpose on the computation an algorithm by which the leader can detect that the computation has terminated. The algorithm must satisfy two properties:

(DT1) The leader can detect that the computation has terminated only if it actually has terminated.

(DT2) If the computation terminates, then the leader eventually detects that it has.

To implement the algorithm, the processes use additional *control* messages. Idle as well as active processes may be required to send and receive control messages. I call the two solutions Dijkstra presented the *tree algorithm* and the *ring algorithm*.

4.5.2 The Tree Algorithm

The tree algorithm [Dijkstra and Scholten 1980] assumes that the processes form a connected directed graph, where process *A* can send messages to process *B* if and only if there is an edge from *A* to *B*. The leader is assumed to have only outgoing edges.[12] Initially, only the leader is active. The control messages are acknowledgments (acks), sent in the opposite direction along edges. It is assumed that no messages are ever lost, but there is no assumption that messages arrive in the order they are sent—not even those sent along a single edge.

Define a process to be in a *neutral* state if it is idle, it has received acks for every message it has sent, and it has sent acks for every message it has received. Initially, only the leader is active and no messages have been sent, so every process except the leader is in a neutral state. The algorithm ensures that after the computation has terminated, every process eventually enters a neutral state. Moreover, when the leader is in a neutral state, all other processes are also in a neutral state.

A process leaves a neutral state only by receiving a message. A process in a non-neutral state remembers on which edge it received that message. Let's call that edge the process's *up edge*. We require that a process acknowledge the receipt of every

[12] A leader that receives messages can be simulated by a pair of processes, one acting as a leader that sends only a single message to the second.

message it has received subject to one rule: it must leave one message received on its up edge unacknowledged until acknowledging it makes the process neutral. In other words, a process can send the one remaining unsent ack to its up process only when it is idle, it has received acks for every message it has sent, and it has sent acks for every other message it has received. It must eventually send that last ack when allowed by the rule.

When process A sends a message to process B that makes the edge from A to B the up edge of B, process A can't become neutral until process B does, because only when B becomes neutral will it send the last ack that A is waiting for from that edge. This implies that the non-neutral processes form a tree with the leader as root, the parent of each non-neutral process B in the tree being the process that is the source of the up edge of B. This in turn implies that if the leader is neutral, then all processes are neutral, so the computation has terminated. Hence DT1 is satisfied, where the leader detects that the computation has terminated when it is in a neutral state.

Once the computation has terminated, all processes are idle and no more messages are sent or received, so all acks except the last one to each non-neutral process's parent will eventually be sent. At that point, any leaf of the tree of non-neutral processes can and eventually must send its last ack and become neutral. Thus, all processes, including the leader, must eventually become neutral, showing that DT2 is satisfied.

4.5.3 The Ring Algorithm

The ring algorithm [Dijkstra et al. 1983] assumes that processes can be numbered from 0 through $N - 1$ such that process number i can send control messages to process number $(i - 1) \bmod N$. The leader is process 0. Control information for the algorithm is also attached to the computation's messages. The algorithm assumes that the computation's message passing is instantaneous. Thus, the atomic action in which process i sends a computation message (with its control information) to an idle process j also makes j active. The algorithm can start with any set of processes active.

As in many distributed algorithms, a process has no control state. Its "code" is a set of *action specifications*, each consisting of a code fragment describing possible atomic actions that can be performed when the code fragment is enabled. The main computation is performed by two kinds of atomic actions of each process i: a *SendMsg(i)* action sends a message from i to another process that at the same time receives it, and a *GoIdle(i)* action puts the process in the idle state.

Termination detection is performed by a series of *probes*, in which a token is passed from each process i to process $(i - 1)\bmod N$, starting from process 0. When

```
process $0$ do
   while true do
      $\langle\, GoIdle(0) \,\rangle$ or $\langle\, SendMsg(0) \,\rangle$ or $\langle\, StartProbe \,\rangle$
   od
od
process $i \in 1\,..\,(N-1)$ do
   while true do
      $\langle\, GoIdle(i) \,\rangle$ or $\langle\, SendMsg(i) \,\rangle$ or $\langle\, PassTok(i) \,\rangle$
   od
od
```

Figure 4.6 The ring algorithm.

process 0 has the token, it can initiate a probe by executing a *StartProbe* action that passes the token to process $N-1$. Each process $i > 0$ can perform a *PassTok(i)* action that passes the token to process $i-1$. The probe ends when the token returns to process 0.

The ring algorithm has the form shown in Figure 4.6. The construct $\langle\, A_1 \,\rangle$ *or* . . . *or* $\langle\, A_k \,\rangle$ indicates that any one of the atomic actions described by A_i can be performed if it is enabled. We now have to define the *GoIdle(i)*, *SendMsg(i)*, *StartProbe*, and *PassTok(i)* actions.

We start by considering what happens if no messages are ever sent, so the only main computation action a process can perform is the *GoIdle(i)* action. In that case, we can let the *PassTok(i)* action be enabled if and only if process i is idle. After the token of a probe started by the leader returns to the leader, every other process is idle and the computation has then terminated when the leader is idle.

This algorithm obviously doesn't work when messages can be sent. After the token is relayed by an idle process, a message from another process can make that process active. When that happens, we allow the token to continue its trip back to the leader, but we ensure that the leader learns nothing from it and must start a new probe. We do this by adding a color to the token. The probe starts with the color white, and it is turned black to tell the leader that the probe has *failed* and it must start a new probe. Process i can make a probe fail by having the *PassTok(i)* action make the token black. Once black, the token remains black for the rest of the probe.

The probe should fail if a process i sends a message to an idle process j that the token has passed. Process j can't make the token black because the token has already passed it, so process i must do it. Our algorithm wouldn't represent a distributed system if we expected process i to know, when it sends the message, whether process j is idle or where the token is. So, we must be conservative. When process i sends *any* message, we have it turn the token black the next time it passes

```
variables $active[i \in 0\,..\,(N-1)] \in \{\textbf{true}, \textbf{false}\}$ ,
          $proclr[i \in 0\,..\,(N-1)] \in \{\text{“white”}, \text{“black”}\}$ ,
          $tokloc \in 0\,..\,(N-1)$ ,
          $tokclr = \text{“black”}$

actions $GoIdle(i)$:    $active[i] := \textbf{false}$

        $SendMsg(i)$:   await $active[i]$ ;
                        with $j \in (0\,..\,(N-1)) \setminus \{i\}$ do $active[j] := \textbf{true}$ od ;
                        $proclr[i] := \text{“black”}$

        $StartProbe$:   await   $(tokloc = 0)$
                                $\wedge ((tokclr = \text{“black”}) \vee (proclr[0] = \text{“black”}))$ ;
                        $tokclr := \text{“white”}$ ;
                        $proclr[0] := \text{“white”}$ ;
                        $tokloc := N-1$

        $PassTok(i)$:   await $(tokloc = i) \wedge active[i]$ ;
                        $tokloc := i-1$ ;
                        if $proclr[i] = \text{“black”}$ then $tokclr := \text{“black”}$ ;
                                                          $proclr[i] := \text{“white”}$
                        fi
```

Figure 4.7 Action specifications of the ring algorithm.

the token. To let process *i* remember that it has sent a message, we add a bit of information that we call the process's color. The color of a process is initially white, and it is made black when the process sends a message. The probe should also fail if process 0 has sent a message, so it should fail if process 0 is black as well as if the token is black.

This line of reasoning has effectively led us to an algorithm. If this were 1983, we would now have to do some careful thinking to find out if that algorithm is correct. Today, such thinking is unnecessary if a model checker can find an execution that shows the algorithm to be incorrect. Instead of thinking, we can write exactly what the algorithm is and run a model checker on it.

A precise description of our algorithm is in Figure 4.7. The variables have the following meanings:

active[*i*] Equals **true** if process *i* is active and **false** if it is idle.

proclr[*i*] The color of process *i*, indicated by the string “white” or “black”.

tokloc The number of the process that holds the token.

tokclr The color of the token.

These variables can have any possible initial value, except that *tokclr* must initially equal “black”. Thus, if the initial state is one in which a probe is being executed, that probe will fail. Here is an explanation of the four kinds of actions:

GoIdle(i) Makes process i idle. The action has no effect (does not change the state) if it is executed when the process is already idle. As observed in Section 4.2.1, an action that has no effect is unobservable, so it makes no difference whether or not it is executed.

SendMsg(i) Enabled if and only if the **await** expression is true—that is, if process i is active. As before, $(0..(N-1)) \setminus \{i\}$ equals the set of all process numbers except i. The **with** statement executes its **do** statement for an arbitrary value of j in that set, making process j active. The action also sets the color of process i to black. (If i is already black and j already active, then the action has no effect.)

StartProbe Performed only by process 0, it is enabled when that process has the token and it or the token is black (or both are black). It makes the process's color and the token's color both white and passes the token to process $N-1$.

PassTok(i) Performed by a process $i \neq 0$, it is enabled when the process has the token and is idle. It passes the token to process $i-1$. If the process's color is black, it makes the token black and itself white. Otherwise, it leaves the process's color white and the token's color unchanged.

A probe is defined to have succeeded if process 0 has the token and is idle, and both process 0 and the token are white—more precisely, when this formula is true:

$$(\mathit{tokloc} = 0) \wedge \neg \mathit{active}[0] \wedge (\mathit{tokclr} = \mathit{proclr}[0] = \text{“white”}).$$

Note that if process 0 and the token are white, then process 0 will not pass the token and the current probe will succeed when process 0 becomes idle.

Any general-purpose model checker should be able to check if the algorithm is correct for small values of N.[13] I would expect an incorrect algorithm to be incorrect even for $N = 3$; and I would be surprised if it were not incorrect for $N = 4$. For $N = 8$, the ring algorithm has 983,806 reachable states and the model checker I used verified DT1 and DT2 in 1 minute, 20 seconds on my laptop.[14]

Having checked that the algorithm is almost certainly correct, I will explain *why* it is correct by writing a correctness proof. I will start by proving the difficult part, DT1, which requires proving that a probe cannot succeed if the computation has not terminated.

[13] A few model checkers cannot check properties like DT2 that assert something must eventually happen.

[14] I checked a slightly modified version of a TLA$^+$ [Lamport 2003] specification of Dijkstra's algorithm written by Stephan Merz, and I also checked (by hand) that the definitions in Figure 4.7 were equivalent to the ones in his specification.

By definition of what it means for a probe to succeed, it suffices to assume that the token has reached process 0 and some process is active, and to prove that the token or process 0 is black. If this is the first time the token reached process 0, then it is black because it was initially black and only process 0 can turn it white. We can therefore assume the token has passed process 0 already, so we are at the end of a complete probe that started at process 0. Since a process passes the token only when it is idle, the existence of an active process means that some process must have become active after it passed the token. Let i be the first process to become active after passing the token. Let j be the process that had the token when i first became active, so $j < i$. Process i can have become active only by receiving a message. Since i was the first process to become active after being passed by the token, every process k with $j < k$ was then idle, so the message that made i active must have been sent by a process k with $k \leq j$. Sending the message made k black, and $k \leq j$ implies that k hasn't yet passed the token. This means that either the token must have been black after k passed it, or else $k = 0$ so process 0 is black. This finishes the proof of DT1.

An examination of the proof reveals that the algorithm remains correct if we modify the *SendMsg*(i) action so it colors process i black only if it sends a message to a process j with $j > i$.

The proof of DT2 is simpler. We assume that all processes are idle and show that a probe must eventually succeed. Since all processes are idle, every process $i > 0$ will pass the token, so the token must eventually reach process 0. If the token and process 0 are both white, then the probe has succeeded. Otherwise, process 0 will start the next probe, which will turn all the processes white. If the token is white when it reaches process 0, that probe will have succeeded. If not, process 0 will then start another probe that must succeed because all the processes are idle and white.

Dijkstra's experience with the garbage collection algorithm tells us that we shouldn't believe these proof sketches. Dijkstra's derivation is more rigorous because he wrote down the invariants implicit in the proof sketches. However, model checking for all values of N from 1 through 8 gives me more confidence in the algorithm's correctness than his derivation. (Neither Dijkstra's derivation nor my proof sketch considers the case $N = 1$.) But this chapter is about algorithms, not about rigorous proofs, so we must be content with a proof sketch.

4.5.4 A Bit of History

In his earlier concurrent algorithms, Dijkstra (and his collaborators) stood alone. No one else had even thought of the problems he was solving. He was also a pioneer in distributed termination detection, but he was not alone. Nissim Francez [1979] was independently working on the same problem and cited an unpublished paper by Michel Sintzoff that was also about the problem.

Dijkstra was careful to cite any work that directly influenced a paper he was writing. For example, the tree algorithm paper cites a lecture by P. M. Merlin for inspiring the problem. However, he did not mention Francez's work, even though he should have been aware of it before Dijkstra and Scholten [1980] was published. It is not clear why he did not follow the usual practice of citing such related work. Perhaps he felt that his termination detection papers were about derivation, not about the algorithms, so Francez's work was not relevant to them.

4.6 Conclusion

The five papers discussed in this chapter are remarkable. The first initiated the field of concurrent algorithms. The second two each started an important subfield of concurrent algorithm research. The last two were "merely" significant. And concurrent algorithms were not Dijkstra's primary interest.

I learned a lot from Dijkstra, and he learned a little bit from me. By the early 1980s, I believe we both felt we had nothing more to learn from each other. I remained cognizant of the debt I and others owed him for his seminal work on concurrent algorithms. However, it was only after he died that I understood how much more we owed him.

There are a number of ideas that have been central to the study of concurrent algorithms—for example, representing the execution of an algorithm as a sequence of states. Many of those ideas were in his papers, sometimes only implicitly. They seemed obvious to me, so I assumed they were well known. But some ideas become obvious only after someone has thought of them. I have come to realize that many of the ideas that we take for granted are due to Dijkstra. Even the idea that one should prove the correctness of algorithms, though not unique to him, was radical at the time.

I have a more personal debt to Dijkstra. Before it ever occurred to me that there could be a science of computing, he was teaching me how to be a computer scientist.

References

E. A. Ashcroft. 1975. Proving assertions about parallel programs. *J. Comput. Syst. Sci.* 10, 1, 110–135. DOI: https://doi.org/10.1016/S0022-00007580018-3.

J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson. 1982. Data requirements for implementation of *N*-process mutual exclusion using a single shared variable. *J. ACM* 29, 1, 183–205. DOI: https://doi.org/10.1145/322290.322302.

P. J. Courtois, F. Heymans, and D. L. Parnas. 1971. Concurrent control with "readers" and "writers." *Commun. ACM* 14, 10, 667–668. DOI: https://doi.org/10.1145/362759.362813.

E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9, 569. DOI: https://doi.org/10.1145/365559.365617.

E. W. Dijkstra. 1968. The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5, 341–346. DOI: https://doi.org/10.1145/363095.363143.

E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inf.* 1, 115–138. DOI: https://doi.org/10.1007/BF00289519.

E. W. Dijkstra. November. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11, 643–644. DOI: https://doi.org/10.1145/361179.361202.

E. W. Dijkstra. 1984. Invariance and non-determinacy. *Philos. Trans. R. Soc. Lond. A* 312, 491–499. DOI: https://doi.org/10.1098/rsta.1984.0072.

E. W. Dijkstra and C. S. Scholten. 1980. Termination detection for diffusing computations. *Inf. Process. Lett.* 11, 1, 1–4. DOI: https://doi.org/10.1016/0020-01908090021-6.

E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11, 966–975. DOI: https://doi.org/10.1145/359642.359655.

E. W. Dijkstra, W. Feijen, and A. van Gasteren. 1983. Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett.* 16, 5, 217–219. DOI: https://doi.org/10.1016/0020-01908390092-3.

R. W. Floyd. 1967. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics, Vol. 19. American Mathematical Society, 19–32.*

N. Francez. 1979. On achieving distributed termination. In G. Kahn (Ed.), *Semantics of Concurrent Computation, Proceedings of the International Symposium*, Evian, France, July 2–4, 1979, Vol. 70: Lecture Notes in Computer Science. Springer, 300–315. DOI: https://doi.org/10.1007/BFb0022476.

D. Gries. 1977. An exercise in proving parallel programs correct. *Commun. ACM* 20, 12, 921–930. DOI: https://doi.org/10.1145/359897.359903.

M. Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1, 124–149. DOI: https://doi.org/10.1145/114005.102808.

D. E. Knuth. 1966. Additional comments on a problem in concurrent program control. *Commun. ACM* 9, 5, 321–322. DOI: https://doi.org/10.1145/355592.365595.

F. T. Krogh. May. 1979. ACM algorithms policy. *Commun. ACM* 22, 5, 329–330.

L. Lamport. 1974a. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* 17, 8, 453–455. DOI: https://doi.org/10.1145/361082.361093.

L. Lamport. 1974b. *On Self-Stabilizing Systems*. Technical Report CA 7412-0511. Massachusetts Computer Associates. https://www.microsoft.com/en-us/research/publication/self-stabilizing-systems/.

L. Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* SE-3, 2, 125–143. DOI: https://doi.org/10.1109/TSE.1977.229904.

L. Lamport. 1986. The mutual exclusion problem: Part II. Statement and solutions. *J. ACM* 32, 1, 327–348. DOI: https://doi.org/10.1145/5383.5385.

L. Lamport. 2003. *Specifying Systems*. Addison-Wesley, Boston. A link to an electronic copy can be found at http://lamport.org.

L. Lamport, R. Shostak, and M. Pease. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst* 4, 3, 382–401. DOI: https://doi.org/10.1145/357172.357176.

S. Owicki and D. Gries. 1976. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM* 19, 5, 279–284. DOI: https://doi.org/10.1145/360051.360224.

5

Origin of Self-Stabilization

Ted Herman

5.1 Introduction

In the year after Edsger Dijkstra received the Turing Award, he turned to a fascinating question. Would it be possible for a distributed system, on its own, to attain correct behavior even if the initial state of machines were incorrect? The setting for this question were his writings, for over a decade, about inter-process communication, operating systems, programming methodology, and program correctness. Within a year, he settled the question affirmatively: self-stabilization, a term he coined[1] for this property, in a distributed system is possible. Edsger first wrote about it in three EWDs, which were condensed and published in a two-page article appearing in the *Communications of the ACM* a year later.

Results of Dijkstra's invention of self-stabilization might seem modest: one short journal article in 1974, a proof presented in a journal in 1986, and some of the previously unpublished reports published in his book [Dijkstra 1982c].

Indeed, there was little uptake of self-stabilization in the computing research community for nearly ten years following the initial publication. Yet, the discovery of computational self-stabilization was acknowledged in the 2002 Influential Paper Award by the ACM Symposium on Principles of Distributed Computing; the annual prize has been renamed to the Dijkstra Award thereafter. Why was this contribution neglected and how did it subsequently gain attention? To answer, we will explain how Edsger's invention of self-stabilization connected to his other interests and consider how the field evolved.

This chapter presents brief technical descriptions of Dijkstra's self-stabilizing algorithms. The EWD articles about them reveal more of Edsger's thought process than does the journal publication [Dijkstra 1974]. His definitions and arguments about correctness are interesting, so I think it worthwhile to show some of them.

[1] Previous to Dijkstra's investigations, self-stabilization was known to be a property of physical or chemical systems.

Later in the chapter I recall application of the idea, how other researchers followed up on self-stabilization, and conclude with some reflections on its significance.

5.2 Privileges in a Ring

We start with the most well known of the self-stabilizing examples because it is one of the jewels of distributed computing. The following lines refer to a cyclic arrangement of $N+1$ processes, with constant $K > N$. (The condition $K > N$ specified here simplifies the presentation below; it can be relaxed to $K \geq N$, which is explained in Chapter 4.) Let $P.i$ refer to process i, for $0 \leq i \leq N$. Each $P.i$ has a single integer variable $S.i$, which can be read by one other process.

$$\begin{array}{llll} S.0 = S.N & \longrightarrow & S.0 := (1 + S.0) \bmod K \quad \square & \text{(a)} \\ (\square\ 1 \leq i \leq N :\ S.i \neq S.(i-1) & \longrightarrow & S.i := S.(i-1)\,) & \text{(b)} \end{array}$$

Notation using the $\square$ operator is from Dijkstra's later work [Dijkstra 1975] defining a guarded command language. The $\square$ denotes nondeterministic selection among statements. Line (b) uses closed form to describe N statements (for $1 \leq i \leq N$). Whereas line (a) is the guarded assignment for process $P.0$, line (b) specifies guarded assignments of the other processes. The program thus consists of one guarded assignment per process. Communication between processes is unidirectional around the ring: let $j = (i+1) \bmod (N+1)$; according to lines (a) and (b), $P.j$ reads $S.i$ and writes $S.j$.

The $N+1$ rules, or guarded assignments, of (a)–(b) abstract the code of a nonterminating system. In other words, system execution consists of repeated selection and evaluation from the set of guarded assignments. For the program above, each guarded assignment's operation is considered to be one atomic step. We assume fairness, that is, for an infinitely long execution, the guarded assignment for each process is selected infinitely often. The notion of fair, nondeterministic scheduling is a way to model arbitrary interleaving of atomic steps among concurrently running processes.

In a situation where the guard for $S.i$ is enabled, we say that $P.i$ is *privileged*. The remarkable property of the program is this: no matter how $S.i$ variables are initialized, under repeated fair application of the rules, eventually only one process is privileged; this despite the lack of explicit detection and correction of what might be considered invalid initial states, for example, initial states where more than one process is privileged. Moreover, in a state with exactly one process privileged, all successive states also have this property; the condition of having exactly one privilege is stable.

What does it mean for a program, algorithm, or system to be self-stabilizing? Intuitively one thinks of phenomena found in physical and biological science, such as systems where order emerges out of chaotic initial conditions. In computing, the system is represented by variables, a program, and a scheduling regime.

- We need some definition of what a system should "stabilize to," or what will be an acceptable *legitimate* behavior established by self-stabilization. Such legitimate behavior could be specified by a list of desired properties, or more formally, using predicates on variable values (possibly involving temporal logic operators).
- Typically, we require an *invariant*; from such an invariant and rules of computation, legitimate behavior can be deduced. An invariant is helpful because once legitimacy is established no subsequent step of computation deviates. Any state satisfying this invariant is called a *legitimate state*. One crucial difference between a self-stabilizing program and an ordinary program is that conditions on the initial state are omitted for the stabilizing program.
- Given any initial state, we require that computation steps establish the invariant. In other words, the precondition for a self-stabilizing system is *true* (there is no specification of initial values). In a distributed system, there can be constituent processes with varying rates of execution, or nondeterministic scheduling of rules. Our reasoning about establishing the invariant has to take such factors into account.

It might seem there is a trivial solution to the second requirement, namely, to have the program for each process first assign safe values to all variables. However, for code to initialize variables (a "first step"), we would need a program counter or the equivalent. Program counters are themselves variables, which have no defined initial value for self-stabilization.

If we assume an ongoing execution of all components of the system, then the individual programs need to be organized as loops. Such loops would perpetually inspect values of variables, correcting their values if something is amiss. In fact, guarded assignments (a)–(b) have exactly this organization: they are perpetually scheduled. For a distributed system, implementation involves communication because directly comparing variables at different locations of the system is not allowed in low-level steps, yet the legitimate-state predicate could specify a relation between distant variables.

Legitimate states for privilege circulation can be described by a regular expression. The legitimate-state invariant is depicted by the set of strings $((m+1) \bmod K)^p m^{N+1-p}$, each string listing values of $S.0$ through $S.N$. Observe that $P.0$ is privileged when $p \in \{0, N+1\}$ (because all $S.i$ have the same value). With $1 \leq p \leq N$, process $P.p$ has the sole privilege.

To prove that every execution of the system ends up with the invariant satisfied, there are several properties that follow from (a)–(b). First is that in any state at least one guard is enabled (shown by contradiction). Second, we reason that if some sequence of steps includes only $P.i$ among $i > 0$, then eventually the system state is legitimate: for all assignments of (b), processes only copy from their predecessors, so all $S.i$ variables end up with the same value (the resulting system state has the form m^{N+1}, which is legitimate). A corollary property is that, in an infinite execution, (a)'s guard is enabled and $P.0$ executes its assignment infinitely often. In fact, in an infinite execution, variable $S.0$ takes on each value in the domain $[0, K-1]$ infinitely many times. Third, observe that only the guarded assignment (a) is capable of introducing a new value to the set of $S.i$ variables. In any initial state, because $K > N$, some value in the domain $[0, K-1]$ is not equal to any of $S.i$ for $i > 0$. Therefore, in an infinite execution, eventually the assignment of (a) must result in $S.0 = m$ such that m is unique among all variables in the ring. Finally, by an inductive argument, the state m^{N+1} is eventually obtained, which is legitimate.

For the program given by (a)–(b), connection to mutual exclusion is by associating "privilege" with access to some shared critical resource since the legitimate-state invariant excludes two or more processes having enabled guards concurrently. Two other standard obligations, absence of deadlock and absence of starvation,[2] are not hard to prove. Specialists might object to classifying such a system as mutual exclusion because, between turns where $P.i$ is privileged, all other processes must be privileged. Properly speaking, the system is *privilege circulation*.

To further appreciate this elegant gem, it is useful to contrast the statements (a)–(b) with earlier research on mutual exclusion and communication between sequential processes. As early as 1959, Dijkstra considered the problem of mutual exclusion framed in terms of atomically accessed shared variables,[3] a challenge met by Dekker's solution [Dijkstra 1962]. At a level close to hardware, tricky situations of program counters and the management of multiple shared variables led to complicated arguments about correctness, avoidance of deadlock, and absence of starvation. The statements (a)–(b) abstract away program counter details and put in clear focus the fundamental problem of overcoming initial inconsistencies, and without reliance on a separate correction mechanism.

[2] In a few mutual exclusion algorithms, it can be that some processes repeatedly enter and exit the critical section, so there is no deadlock, yet other processes never enter their critical section, a situation known as starvation.

[3] The history is documented in Dijkstra [1987a].

The first write-up of self-stabilizing privilege circulation is Dijkstra [1973a], and it is my own favorite Dijkstra's paper on the topic. He wrote this about the discovery:

> [T]he problem has been with me for many months, while I was oscillating between trying to find a solution—and many an at first sight plausible construction turned out to be wrong!—and trying to prove the non-existence of a solution.

The narrative arc of Dijkstra [1973a] proceeds from stating desired properties to a near derivation of the result by explaining design choices and influences. One consideration given in Dijkstra [1973a] explains why rule (a) needs to be different from rule (b): if $N + 1$ is not a prime number, then the ring can be composed of segments so that there is a privilege in each segment, and an adversarial (yet fair) schedule will have multiple privileges forever moving around the ring. By such an initial state and constructed schedule, it is impossible to arrange one uniform rule for all processes if we require stabilization to a single privilege.

I had not met Edsger during the time he solved the question of self-stabilizing distributed systems. I came to know him, only as a graduate student, at UT Austin, some ten years later. However, I was able to discuss the topic with him. I feel confident that some private feelings of delight accompanied his discovery of a solution.

5.3 Privileges in an Array

The solution (a)–(b) for privilege circulation has $N+1$ counters, each representable with $O(\log N)$ bits. One might ask whether or not self-stabilizing privilege circulation is possible for a system of $N + 1$ processes with a constant number of bits per process. Dijkstra answered this question affirmatively in two notes: Dijkstra [1973b, 1973c]. The former presents a solution using four states per process, and the latter shows a system with three states per process. We cover the four-state solution here and omit describing the three-state case because both have similar characteristics. The three-state version [Dijkstra 1973c] is a model of concise presentation along with proof obligations, later polished and published in Dijkstra [1986].

In Dijkstra [1973b], the four process states are taken from $\{0, 1\} \times \{up, down\}$ and the two endpoints of the array are named *bottom* and *top*. Rather than precisely following notation of Dijkstra [1973b], my abbreviated presentation denotes a process state by Xd where $X \in \{0, 1\}$ and direction $d \in \{\uparrow, \downarrow\}$, thus using ↑ for *up* and ↓ for *down*. The system consists of $N + 1$ processes with each process accessing states of its neighbor variables: the array is oriented vertically so that process 0 is *bottom*, process N is *top*, and the remaining $N-1$ processes each have two neighbors, below and above. An important stipulation for *bottom* and *top* is that direction for them is immutable: the form is $X{\uparrow}$ for *bottom* and is $X{\downarrow}$ for *top*. To make the presentation more concise, we define $\sim X \equiv (1 - X)$.

Process	Guard			Assignment result
bottom: $i = 0$		$X{\uparrow}$	$X{\downarrow}$	$\sim X{\uparrow}$
top: $i = N$	$\sim X?$	$X{\downarrow}$		$\sim X{\downarrow}$
$0 < i < N$		$X{\uparrow}$	$X{\downarrow}$	$X{\downarrow}$
$0 < i < N$	$\sim X?$	$X{\downarrow}$		$\sim X{\uparrow}$

The table lists processes, corresponding guards, and the resulting assignments. Each guard is a pattern that lists successive values for the below process (if it exists), the process state, and the above process (if it exists), for example, the first row is missing a below process because the rule is for *bottom*, the second row is missing an above process because the rule is for *top*. The third row, for middle processes, is missing a below term because the guard only refers to the process itself and the neighbor above (the fourth row similarly misses an above term). The center sub-column under the Guard heading specifies the state of process i: it is the only value that changes by assignment, provided that the guard is enabled and the rule is selected by the schedule. The center sub-column under the Assignment result heading shows the effect of changing the state of process i. The first row of the table indicates that *bottom* reads the state of the process above; if they have the same X value and their direction modalities differ, then the guard is enabled, and the assignment changes only the X value for *bottom*. Some guards have "?" for the direction, indicating a wildcard for pattern matching. Inspecting the table, we see that middle processes each have two rules; however, the patterns of their guards are exclusive. Typical legitimate states, by way of regular expressions, include these forms:

$$\text{privilege moving up} \quad (X{\uparrow})^k(\sim X{\downarrow})^{N+1-k}$$

$$\text{privilege moving down} \quad (X{\uparrow})^k(X{\downarrow})^{N+1-k}$$

As the privilege moves up, both X and direction change, whereas with the privilege moving down, only the direction changes. (Aside: I first saw the idea of using regular expressions to describe the state on the blackboard when Edsger was working out some idea during a Tuesday afternoon club meeting.)

A proof of correctness, that is, that requirements itemized in Section 5.2 are met, rests on observations about how privileges move. First, it needs to be shown that no state of the system is without privileges; second, that the number of privileges cannot increase; third, that a state with more than one privilege leads, by execution of the rules, to a state with fewer privileges; and finally, with one privilege, that the subsequent execution circulates that privilege.

To give just an informal flavor of the proof, we outline one of the arguments. That a privilege exists in any state is shown by supposing all guards *false*, then establishing a contradiction. Looking at the fourth rule in the table, we see that the guard would be enabled at any middle process if X values differ along the array, hence all X values must be equal, save possibly the top process; then by the second rule, the top process must have the same X as all the others. By similar reasoning we infer that the direction of all processes, except *bottom*'s, is $\downarrow$, which contradicts the falsehood of the first rule's guard.

5.4 Distance Convergence on a Circle

The first mention of self-stabilization is from a trip report dated August 7 [Dijkstra 1982a] and a solution dated the day after: "The problem solved in this note arose in connection with the (just initiated) study of self-stabilizing systems." [Dijkstra 1982b]. The first privilege circulation paper [Dijkstra 1973a] is dated about two months later. Yet Edsger's first-written exploration had nothing to do with systems or privileges.

The problem formulation starts with an arbitrary placement of $N > 2$ points on a circle of unit circumference and a rule whereby points move their positions on the circle. The question proposed is whether or not repeated application of the rule, described in the next paragraph, is a convergent process; that is, do point positions stabilize within a finite number of rule applications?

Let $x.i$ denote a point on the circle, for $0 \leq i < N$. Consider the clockwise shortest path from $x.i$ to $x.(i+1) \bmod N$, denoted as path $p.i.(i+1)$. The length of this path, thanks to definition of a unit circle, is a real-valued number in $[0, 1)$. Similarly, $p.(i-1).(i+1)$ is the shortest clockwise path from $x.((i-1) \bmod N)$ to $x.((i+1) \bmod N)$, whose length lies in $[0, 1)$. The rule of update is this: move $x.i$ to the midpoint of path $p.(i-1).(i+1)$. This rule is idempotent in the sense that if $x.i$ is already at the midpoint between $x.(i-1)$ and $x.(i+1)$, then application of the rule changes nothing.

Let $d.i.(i+1)$ denote the length of $p.i.(i+1)$. A postcondition of any assignment to $x.i$ is $d.i.(i+1) < \frac{1}{2}$, $d.(i-1).i < \frac{1}{2}$, and that both of these distances are equal. Consider now an update to $x.(i-1)$ or to $x.(i+1)$, which could change $d.(i-1).i$ or $d.i.(i+1)$, possibly by increase or decrease. Fortunately, the postcondition of assignment on either side of $x.i$ maintains the bound on neighbor path distances: each remains below $\frac{1}{2}$. The generalization of this observation is that, if the update rule is applied to all N points, then invariantly all neighbor distances remain below $\frac{1}{2}$ in any subsequent rule application.

At this point, we are tempted to think that convergence follows readily. Dijkstra introduced a metaphor: "The system converges linearly (imagine successive points connected by spiral springs or rubber bands of equal length)." In fact, without a crucial assumption, convergence is not guaranteed. The surprise is that if the rule of update is applied to all $x.i$ atomically, in parallel, then the system fails to converge. In Dijkstra [1982b], two simple scenarios, for $N = 3$ and $N = 4$, demonstrate that distances can oscillate forever if parallel update is permitted. By this observation about the rule failing to converge, we recognize a theme repeated in the literature of distributed computing: changes to scheduling assumptions can make the difference between an algorithm being correct and incorrect. In other investigations years before, Dijkstra had also to reckon with scheduling, concurrency, and atomicity when he dealt with problems of semaphores, producer–consumer situations, and dining philosophers.

A sufficient condition for convergence of the points on a circle is central, fair scheduling of the rule, nondeterministically updating one point in each step; such scheduling is fair if no point is ignored permanently. Another way to present the system is to think of each point as a process that operates by atomically reading positions of its neighbors and updating its own position. Some years after the fair scheduling abstraction was introduced, it became a controversial topic. In fact, Dijkstra later changed his mind and wrote a critical position paper on the topic [Dijkstra 1987b].

Fairly scheduled, all points stabilize to the predicate that neighbor distances are below $\frac{1}{2}$. What then of further convergence, say stabilization to all distances eventually remaining constant? Notice here that the sequence of points in clockwise order, going from $x.0$ to $x.(N-1)$, may go around the circle a number of times. It is not necessary to reason about the circle once all neighbor distances are bounded by $\frac{1}{2}$: let us consider a graph with $\{x.i \,|\, 0 < i \leq N\}$ as vertices and an edge $(x.i, x.((i+1) \bmod N))$ having weight $d.i.(i+1)$. The corresponding update rule for $x.i$ in the graph is to replace neighbor weights by their average. The sum of all weights thus remains unchanged by any update.

One consequence of an update for $x.i$, in the case where $x.i$'s neighboring edge weights differ, is this: the larger of the weights becomes smaller. Let f be the N-tuple constructed from a descending sort of the edge weights; for a rule update that changes weights, with f as the tuple before the update and f' as the tuple after, we have $f' < f$ as a variant function in the sense of lexicographic comparison. The system terminates when all weights are equal. This variant function does not prove finite termination, it only gives us convergence in the limit because weights are real numbers. The final exploration of Dijkstra [1982b] looks at replacing real-valued distances with rational numbers that are multiples of $1/p$ for some $p \gg N$ and then using a round-off technique to enforce a finite convergence.

5.5 Complexity and Verification

The stated goal of Dijkstra [1973a] was to answer the question of *possibility* of a self-stabilizing system. Investigation of its complexity was not in the scope of these initial works. Nonetheless, Dijkstra did work on reducing state complexity, for example, constant-size state instead of $O(\log(N))$ state for privilege circulation, and Dijkstra [1973b] does state an $O(N^2)$ bound on the minimum number of state changes to reach a legitimate state. The continuation of research by others defined other complexity metrics on stabilization time, the overhead of self-stabilization, and so on. Section 5.8 describes some ways subsequent research investigated complexity.

In the years after Dijkstra [1973a], the specific task of verifying self-stabilization gained in terminology and technique. The general recipe is to (*i*) define what is a legitimate state and show that it is invariant with respect to any rule application; (*ii*) analyze execution of rules from an illegitimate state to reason that, eventually, the system does reach a legitimate state. These two parts are called *closure* and *convergence* in Arora and Gouda [1993], the latter technically achieved using some bounded variant function of states.

Recall the structure of the analysis of points on a circle. One first observes there an initial phase of stabilization so that all distances are less than $\frac{1}{2}$, then one reasons about further convergence of the system. In effect, a weaker invariant can be a stepping-stone to the legitimate-state invariant, which is a helpful way to structure the proof of convergence. This idea was used often in subsequent literature of self-stabilization. It is called the *staircase* method in Gouda and Multari [1991] and applies to modular constructions of self-stabilizing systems.

A methodological cousin to the staircase method is a compositional approach to building self-stabilizing algorithms. Suppose A and B are both self-stabilizing, B is allowed to read variables of A (but not *vice versa*), and execution of the combination of A and B is fair; then the composition $A||B$ is self-stabilizing. Compositionality is generally attractive in system engineering. Parallel composition of stabilizing algorithms was also investigated in Dolev and Herman [1999].

5.6 Relevance

In the introduction of his self-stabilization article, Dijkstra said little on what it might be used for, or how and where it might be applied. From Dijkstra [1973a] we have this:

> Whether the property of self-stabilization is interesting either as a start procedure, or for the sake of system robustness or merely as an intriguing problem, is a question that falls outside the scope of this article.

In that quote, we see a glimmer of systems dealing with faults, and indeed that was the predominant flavor of subsequent research in the distributed computing community. Whereas the definition in Dijkstra [1973a] defines the property by "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps," subsequent research suggests the role of intermittent failures. So-called *transient faults* corrupt mutables (variables, registers, channels, and so on) arbitrarily, followed by a period of stability long enough for convergence to legitimate behavior. That view classifies self-stabilization as a technique for recovery from transient faults. Each occurrence of a transient fault results in a new, possibly corrupt, initial state.

There are different schools of thought on how to deal with faults. Transient faults in memory and processors are now common due to *soft errors* [Li et al. 2016]; however, self-stabilization is not used there because effects of errors would be visible during convergence to a legitimate state. Instead, by engineering redundancy and exploiting low-level hardware properties, transient fault impact is, with high probability, contained at the firmware layer or below. At higher layers there can be timing faults and communication lapses that cannot be *masked* (hidden); one idea to live with such lapses is *eventual consistency* [Vogels 2009], where behavior or data become consistent if there is a long enough period without faults—a technique similar in spirit to self-stabilization's accommodation of transient faults. In an asynchronous message-passing system, even if there are no transient faults that directly hit process states, it can be possible that a sequence of node crashes and restarts, together with faults in message channels, essentially drives processes to erroneous states, motivating self-stabilization's assumption of arbitrary initial states [Varghese and Jayaram 2000].

Section 5.8 lists some specific ways that research has further progressed on issues of fault tolerance. Generally, if informally, the community recognizes that the type of error recovery offered by self-stabilization can be appropriate for some problem domains and inappropriate for others. A practical example here is the Address Resolution Protocol used in local area networks to translate interface addresses into IP addresses. Each host in the network maintains a table for this purpose and periodically refreshes entries in the table by message exchanges; older table entries are discarded. After transient faults, tables could have corrupt values. If we presume the protocol self-stabilizes, then during the period of convergence some or all users of the network will be unable to communicate: this is inconvenient but usually tolerable. A different example is a replicated, distributed database. If a transient fault were to corrupt data or inject arbitrary values, then even after convergence to a state where all consistency relations on the data are satisfied, thus legitimate according to the database schema, the persistence of faulty data would not be acceptable. The convergence of self-stabilization would not be appropriate here. Preferable for databases is rollback to a known consistent state with valid data.

5.7 Adoption

Following publication of the original self-stabilization article [Dijkstra 1974], there was very little reference to the concept in computing science research. I think this publication was overlooked for several reasons. The paper does not stress where self-stabilization might be applied; the field of distributed systems had not yet advanced to the scale where self-correcting systems were seen as important; and the brief presentation using an abstract model of computation did not spell out details in familiar terms. While programming by a set of rules is familiar in logic programming or term-rewriting systems, it is uncommon in distributed systems.[4]

One paper I need to quote is Kruijer [1979], because it generalizes privilege circulation in a direction that intersects with another of Dijkstra's interests. Recall that the four-state solution stabilizes to having one privilege circulate from bottom to top process, and back. Informally, the generalization is to expand the topology from an array to a tree; thus the top process, for example, becomes the root of the tree, each interior process can have multiple children, and instead of a single bottom process, we have numerous leaf processes. Rather than a single privilege, guards that propagate a privilege upward are enabled only when all children have privileges. What we get is a self-stabilizing wave behavior, launched from the root, reflected by the leaves, and coalescing back to the root. A similar idea is found in Dijkstra's exploration, with his longtime collaborator Carel Scholten, of *diffusing computation* [Dijkstra and Scholten 1979] and termination detection.

[4] One recent paper highlighting the potential of rule-based programming for fault-tolerant systems is Stutsman et al. [2015].

I could find but three published papers citing Dijkstra [1974] up through 1983,[5] but in that year an event in the resurrection of self-stabilization took place. Leslie Lamport gave an invited presentation at the ACM Symposium on Principles of Distributed Computing,[6] where he challenged the audience by asking how many of them had read Dijkstra [1974] (most people responding had not). Lamport said "I regard this as Dijkstra's most brilliant work" and he went on to say "I regard self-stabilization to be a very important concept in fault tolerance, and to be a very fertile field for research." In Chapter 4, Leslie Lamport refers to his invited talk and gives his own perspective on the story of self-stabilization. Research based on Dijkstra [1974] picked up in the 1980s; a workshop dedicated to self-stabilization began in 1989, continued in five other years, and eventually became the Symposium on Stabilization, Safety, and Security of Distributed Systems that has continued annually up to the present.

5.8 Impact

After Dijkstra's foundational work of 1973, a considerable literature on self-stabilization continued through the efforts of others. This chapter barely touches on topics found in this literature: this is by no means a comprehensive survey. To date, there are two books dedicated to self-stabilization [Dolev 2000, Altisen et al. 2019], chapters in books [Tel 2000, Ghosh 2006, Tixeuil 2009, Kshemkalyani and Singhal 2011], journal papers, and many conference articles related to distributed systems. I have organized some of the common research themes under headings in the remainder of this section.

5.8.1 Possibility

Some research questions flowed from Dijkstra [1973a, 1973b]. One example is the impossibility of a *uniform* solution to privilege circulation in a ring (uniformity meaning that all processes have the same rules), Dijkstra's impossibility proof was based on $N+1$ being non-prime. In two directions, papers showed a uniform solution is possible: if $N+1$ is prime [Burns and Pachl 1989] or if the solution uses randomization (which breaks a cycle of illegitimate states) [Herman 1990]. Another example of work that continues from Dijkstra [1973a] is the already mentioned adaptation of the four-state solution to the tree topology [Kruijer 1979].

Before describing other results, let us recall that Dijkstra's first problem solved, points on a circle, has a fixed-point legitimate state rather than privilege

[5] One paper building on the theme of self-stabilizing mutual exclusion and citing Dijkstra [1974] is Tchuente [1981].

[6] Transcript was published the following year [Lamport 1984].

circulation. In general, self-stabilization programs are of two kinds—fixed point solutions or specific intended behaviors. A hybrid of these kinds would be a behavior for which certain output variables reach a fixed point, whereas other variables thereafter continue changing after convergence. This distinction is behind the definition of *silent stabilization* which specifies that, eventually, all communication variables between processes do not change. In Dolev et al. [1999], it is shown that for selected problems silence is impossible under a memory-size constraint.

The trend of results on possibility and impossibility of self-stabilization is similar to the examples above: whether a given task is possible depends on differences in computing models, scheduling assumptions, or other specialized constraints. Generally, researchers found that self-stabilization is possible in theory for large classes of problems. Some papers proposed mechanisms to transform non-stabilizing algorithms into stabilizing versions [Afek et al. 1990, Awerbuch and Varghese 1991, Katz and Perry 1993].

Several computing tasks are, by definition, impossible to solve directly by self-stabilization. The best-known example is consensus, where each participating process can write only once to its decision variable. An initial state that puts the program counter at an operation to write the decision variable with an incorrect value defies any attempt to reach a legitimate state. Faced with this impossibility, the natural approach is to look for a system that repeatedly uses consensus. Thus, researchers use a weakened form of consensus that tolerates some initial incorrect decisions but eventually converges to desired behavior. For example, it is possible to employ standard components of distributed consensus, making self-stabilizing versions of them, and then use composition to assemble the components into a system [Angluin et al. 2006, Dolev et al. 2010].

5.8.2 Applications

The initial publication on self-stabilization was concerned with privilege circulation, inspired by mutual exclusion. It is natural to ask which other problems of system control admit self-stabilizing solutions. Many papers are in the category of distributed system control, showing the possibility of self-stabilizing synchronizers, phase synchronization, local mutual exclusion, and wave behavior.

Distributed systems typically require network protocols, and self-stabilization is useful at various layers in network architecture: packet-window protocols, routing algorithms, overlay networks, and peer-to-peer systems all have self-stabilizing results. Especially popular are self-stabilizing algorithms treating the communication topology as a graph, which leads to standard graph problems of finding an independent set, shortest path, coloring edges, coloring vertices, and so forth.

Finding a rooted, spanning tree, for example, is one way to solve leader election, which can then be used as a tool to solve other tasks in a self-stabilizing manner.

5.8.3 Complexity

Complexity has numerous meanings for self-stabilization. The usual measure considered in research is the time for convergence, and it can be examined in different ways. One measure is to count the number of state changes or number of steps (ignoring "stuttering" steps that do not change state). However, if processes can execute concurrently, then counting steps could be an overestimate of the convergence time. Some authors advocate counting the number of *rounds* to reach a legitimate state. Informally, a round is a collection of steps in which each process has a turn at making progress. A further complication is that fair scheduling, which models the nondeterminism resulting from arbitrary process computing speeds, skews the relation between time and steps or rounds. Papers in the literature typically bound convergence complexity for the worst case of an initial state and scheduler choice; one technique is to use a variant function to prove termination (of convergence) and then analyze that function to derive a bound. In distributed systems implemented by message passing, further complexity measures could count the number of messages needed for convergence, the size of messages, and related details.

Is worst-case convergence complexity the right measure for self-stabilization? One idea is that, although self-stabilization guarantees convergence from an arbitrary initial state, the usual case for a transient fault is that relatively few system components (variables or processes) are affected by a fault. Therefore, it can be interesting to measure convergence complexity from k-fault initial states (in which at most k components have damage) [Kutten and Peleg 1995, Genolini and Tixeuil 2002, Datta et al. 2013]. Another idea is to measure average-case complexity (especially for randomized self-stabilizing algorithms) [Fernández-Zepeda and Alvarado-Magaña 2008, Fallahi et al. 2013].

When Dijkstra's investigation moved from privilege circulation in the ring to an array, he reduced the size of variables from a function of N to a constant. Similarly, we can use the size of the state-space of a program as the evaluation of its space complexity. Some papers focus on self-stabilizing solutions to problems that have small (possibly optimal) space complexity [Herman 1992, Beauquier et al. 1999, 2007, Blin et al. 2014, Blin and Tixeuil 2018, Sudo et al. 2018].

Another aspect of complexity could be the overhead of stabilization compared to systems that are not stabilizing. For a distributed system that exhibits a desired behavior, one could compare the performance between stabilizing and non-stabilizing versions, which is only sensible for executions that begin in non-corrupt initial states (because a non-stabilizing version's execution could remain forever in illegitimate states).

5.8.4 Atomicity

It wasn't for some years after Dijkstra [1974] that axioms and terminology for atomicity of shared memory were defined [Lamport 1979, 1986a, Misra 1986]. Using such tools, researchers considered self-stabilization in an architecture where processes communicate through shared registers, typically single-writer and single-reader registers (the book Dolev [2000] is a place to learn more). Local computations within processes can thereby be fully parallel because the only synchronization is by reading or writing registers. The architectural perspective here is of finer granularity than the model of guarded commands: whereas a guarded command could read and write to multiple variables in one step, registers are less powerful. Consequently, researchers have revisited questions of possibility in the more granular register model of computation [Dolev et al. 1993, Ducourthial and Tixeuil 2001].

5.8.5 Schedulers

Dijkstra [1973a] described two schedulers, personified as demons. The case of "centralized control" has a demon standing in the center of the ring which sends processes, one at a time, a command to adjust themselves (according to the applicable rule from (a)–(b)). The other case Dijkstra called the distributed demon. Chapter 4 discusses how privilege circulation in the ring differs between these schedulers. A general analysis of different kinds of demons in self-stabilization research is presented in Dubois and Tixeuil [2011]. Scheduler behavior can be constrained by various parameters, including fairness, concurrency, sensitivity to whether a guard is enabled, and so forth. For analysis of convergence, bounding worst-case complexity depends on which scheduler is chosen; some authors describe the scheduler as an adversary that perpetually chooses the rule or process in a way to delay convergence as long as possible.

For reasoning about programs, it can be more convenient to assume fairness and coarse atomicity (such as centralized control) compared with the situation without fairness and when atomicity is fine-grained. Accordingly, the problem of simulating one demon by another has been studied by researchers. For example, it is possible to automatically convert a self-stabilizing program based on centralized control into one that only requires the distributed demon [Dubois and Tixeuil 2011].

5.8.6 Model Exploration

In the spirit of the question about what problems admit self-stabilizing solutions, researchers expanded the scope of self-stabilization by looking at other computing models. Early efforts showed stabilizing algorithms for message-passing communication rather than the shared variable type of communication between processes [Afek and Brown 1993, Varghese 2000]. At around the same time, some papers specified programs in a traditional Von Neumann architecture of program counters [Lamport 1986b, Dolev et al. 1990]. Other settings for stabilization research include synchronous parallel execution [Herman 1990], overlay networks [Feldmann et al. 2020], sensor networks [Herman 2003, Herman and Tixeuil 2004], and sequential operations on a data structure [Herman and Masuzawa 2001].

Recall that the first problem Dijkstra solved, points on a circle, called for convergence to a fixed point rather than privilege circulation. Subsequent research by others returned to problems with fixed point legitimate states, such as graph-theoretic algorithms, including matching [Hsu and Huang 1992], coloring [Gradinariu and Tixeuil 2000], spanning tree formation [Arora and Gouda 1994], and leader election [Blin and Tixeuil 2020]. A surprising likeness to Dijkstra [1982b] is found in the abstracted problem of mobile robot formation [Suzuki and Yamashita 1999, Défago et al. 2006, Flocchini et al. 2008, 2012], where autonomous robots are points in space and their task is to collectively achieve a formation. Some of these robot algorithms are self-stabilizing [Dieudonné and Petit 2012, Ooshita and Tixeuil 2015, Défago et al. 2019, 2020].

More recently, a number of articles associate self-stabilization with population protocols, a mobility-inspired model that emphasizes convergence [Aspnes and Ruppert 2009]. Agents have local state and meet in pairs, communicate with each other, and change their state. The schedule of meetings can be selected by an adversary, subject to constraints, for example, that all pairs are selected fairly, or randomly. One challenge is to show eventual convergence, but with limited state size, and in the case of self-stabilization, with arbitrary initial state [Angluin et al. 2008].

5.8.7 Fault-tolerance

The study of fault-tolerant algorithms and systems is a core topic of distributed computing. Though self-stabilization addresses transient faults, one can consider further aspects of traditional tolerance, including Byzantine failures, fail-stop processes, or in the case of message-passing models, spurious injection of messages, message duplication, and so on. For example, it is possible for a system to combine the forward-recovery technique of self-stabilization with sufficient replication to tolerate a limited number of Byzantine processes [Nesterenko and Arora 2002, Dolev and Welch 2004, Masuzawa and Tixeuil 2006]. The question of space locality for Byzantine convergence in stabilization is investigated in Dubois et al. [2012, 2015]. The combination of self-stabilization and classical error detection/correction was studied in [Herman and Pemmaraju 2000, Gouda et al. 2006].

Two specializations of self-stabilization are fault containment [Ghosh et al. 2007, Köhler and Turau 2012] and snap-stabilization [Cournier et al. 2002]. A program is fault-containing if, in addition to being self-stabilizing, it has the further property that if a legitimate state is perturbed at only a single process, then within $O(1)$ time the behavior of the program becomes legitimate. Snap-stabilization is concerned with call–response problems, such as operations on a distributed data structure or initiation and termination of a wave algorithm. The property required for snap-stabilization is that any *new* call, following a transient fault, is guaranteed to have a correct result and response.

One way to cope with the complications of details in the computing model is to limit the scope of transient faults. To purists, a transient fault means that everything from registers to program counters, message buffers, and any variable could be corrupt: only the code itself remained untouched. Dijkstra only expressed ideas at a higher level; there is no reason self-stabilization could not be useful for more benign definitions of transient fault. For example, self-stabilization is useful when select services of a distributed system are made self-stabilizing, for example, clock synchronization, even though the system as a whole is not stabilizing. Another example is the self-stabilizing mutual exclusion protocol in Lamport [1986b], which has a traditional program style of a common memory accessed by all processes: communication variables could be initially arbitrary, but rather than reason about program counters and memory pointers, Lamport carefully stated minimal assumptions about code properties that must hold to establish self-stabilization.

5.9 Conclusions

A favorite problem Edsger assigned students was binary search. He asked students, in class, to write down the solution [Dijkstra 1989]. Most students failed to correctly solve the problem, mainly due to corner cases, because they had not *derived* the solution from the specification. A crucial property of a correct program for binary search is that it terminates and finds an index satisfying the search, even when the input array is not sorted. Particularly as a student of self-stabilization, I recall being impressed by this observation of Edsger. The idea of finding new properties enjoyed by an old program can be intriguing.[7] A question near the end of the first EWD on privilege circulation [Dijkstra 1973a] is whether the system is self-stabilizing even if guarded assignments are not entirely atomic (the answer is yes, as Dijkstra showed, in the same EWD).

For readers who are new to the topic, it can seem that self-stabilization is a drastic, possibly wasteful method for dealing with faults. Rather than engineering tolerance for the most likely cases, the method is to suppose the most general, arbitrary kind of transient fault. In fact, this is a well-known tactic for problem solving in mathematics, that is, simplification through generalization. The disadvantage of engineering for specific cases of transient faults can be considerable software complication, whereas by a general approach we take care of all cases. This is an important aspect of Edsger's discovery of the topic.

What was, and remains, the significance of self-stabilization for distributed systems? When I look around today, it doesn't seem that we have such systems in the purest sense of the term. Practitioners managing distributed systems will appreciate the appeal of self-stabilization, but large-scale implementation is elusive. Instead, what was manual response to failure events has been largely automated, using watchdog approaches or failure detectors instead of putting self-stabilization into the design of system services. Arguably the most important lesson from Dijkstra's work has been the notion that systems can be useful even though incorrect behavior is exposed while the system recovers from a fault. Data centers and distributed systems grew to immense scale from the 1990s to the present. One perception by developers of data-intensive applications was this: in the presence of network failures, there seemed to be a conflict between high availability and data consistency [Brewer 2012]. In enterprises with large distributed systems, access to data is important to their business, yet maintaining such access while tolerating network outages and ensuring data consistency between different processes of the system did not seem possible [Brewer 2012]. As a consequence, some engineers studied applications and found that, while some applications require total correctness (safety first), other applications (or their users) are able to tolerate eventually "X" for some definition of X. (The earlier mentioned Vogels [2009] is an example of the latter). Dijkstra's discovery predated other adaptations of eventually correct behavior.

A quip, heard especially in research funding conversations, is that the outcome of any successful project is that more research is needed. By that philosophy, Edsger Dijkstra's project on self-stabilization was superbly successful.

[7] You can try it for yourself, asking what would happen to the solution of Dijkstra [1982b] if one of the points on a circle were "broken" in the sense it could not move.

Acknowledgment

I thank Sébastien Tixeuil for his careful reading of a draft of this chapter, which brought about many interesting observations and suggestions on the selection of references.

References

Y. Afek and G. M. Brown. 1993. Self-stabilization over unreliable communication media. *Distrib. Comput.* 7, 1, 27–34. DOI: https://doi.org/10.1007/BF02278853.

Y. Afek, S. Kutten, and M. Yung. 1990. Memory-efficient self stabilizing protocols for general networks. In J. van Leeuwen and N. Santoro (Eds.), *Proceedings of the Distributed Algorithms, 4th International Workshop, WDAG'90*, Bari, Italy, September 24–26, 1990, Vol. 486: Lecture Notes in Computer Science. Springer, 15–28. DOI: https://doi.org/10.1007/3-540-54099-7_2.

K. Altisen, S. Devismes, S. Dubois, and F. Petit. 2019. *Introduction to Distributed Self-Stabilizing Algorithms*. Morgan & Claypool.

D. Angluin, M. J. Fischer, and H. Jiang. 2006. Stabilizing consensus in mobile networks. In P. B. Gibbons, T. F. Abdelzaher, J. Aspnes, and R. R. Rao (Eds.), *Proceedings of the Distributed Computing in Sensor Systems, Second IEEE International Conference, DCOSS 2006*, San Francisco, CA, June 18–20, 2006, Vol. 4026: Lecture Notes in Computer Science. Springer, 37–50. DOI: https://doi.org/10.1007/11776178_3.

D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. 2008. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.* 3, 4, 1–28. DOI: https://doi.org/10.1145/1452001.1452003.

A. Arora and M. Gouda. 1993. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Trans. Softw. Eng.* 19, 11, 1015–1027. DOI: https://doi.org/10.1109/32.256850.

A. Arora and M. G. Gouda. 1994. Distributed reset. *IEEE Trans. Comput.* 43, 9, 1026–1038. DOI: https://doi.org/10.1109/12.312126.

J. Aspnes and E. Ruppert. 2009. An introduction to population protocols. In B. Garbinato, H. Miranda, and L. Rodrigues (Eds.), *Middleware for Network Eccentric and Mobile Applications*. Springer, 97–120. DOI: https://doi.org/10.1007/978-3-540-89707-1_5.

B. Awerbuch and G. Varghese. 1991. Distributed program checking: a paradigm for building self-stabilizing distributed protocols (extended abstract). In *Proc. 32nd Annual Symp. on Foundations of Computer Science*. IEEE Computer Society, 258–267. DOI: https://doi.org/10.1109/SFCS.1991.185377.

J. Beauquier, M. Gradinariu, and C. Johnen. 1999. Memory space requirements for self-stabilizing leader election protocols. In B. A. Coan and J. L. Welch (Eds.), *Proc. ACM SIGACT-SIGOPS 18th Symp. on the Principles of Distributed Computing*. ACM, 199–207. DOI: https://doi.org/10.1145/301308.301358.

J. Beauquier, S. Delaët, S. Dolev, and S. Tixeuil. 2007. Transient fault detectors. *Distrib. Comput.* 20, 1, 39–51. DOI: https://doi.org/10.1007/s00446-007-0029-x.

L. Blin and S. Tixeuil. 2018. Compact deterministic self-stabilizing leader election on a ring: The exponential advantage of being talkative. *Distrib. Comput.* 31, 2, 139–166. DOI: https://doi.org/10.1007/s00446-017-0294-2.

L. Blin and S. Tixeuil. 2020. Compact self-stabilizing leader election for general networks. *J. Parallel Distrib. Comput.* 144, 278–294. DOI: https://doi.org/10.1016/j.jpdc.2020.05.019.

L. Blin, P. Fraigniaud, and B. Patt-Shamir. 2014. On proof-labeling schemes versus silent self-stabilizing algorithms. In P. Felber and V. K. Garg (Eds.), *Proceedings of the Stabilization, Safety, and Security of Distributed Systems—16th International Symposium, SSS 2014,* Paderborn, Germany, September 28–October 1, 2014, Vol. 8756: Lecture Notes in Computer Science. Springer, 18–32. DOI: https://doi.org/10.1007/978-3-319-11764-5_2.

E. Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2, 23–29. DOI: https://doi.org/10.1109/MC.2012.37.

J. E. Burns and J. Pachl. 1989. Uniform self-stabilizing rings. *ACM Trans. Program. Lang. and Syst.* 11, 2, 330–344. DOI: https://doi.org/10.1145/63264.63403.

A. Cournier, A. Datta, F. Petit, and V. Villain. 2002. Snap-stabilizing PIF algorithm in arbitrary networks. In *Proc. 22nd IEEE Int. Conf. on Distributed Computing Systems*. IEEE, 199–205. DOI: https://doi.org/10.1109/ICDCS.2002.1022257.

A. K. Datta, L. L. Larmore, S. Devismes, and S. Tixeuil. 2013. Fast leader (full) recovery despite dynamic faults. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, Cambridge, MA, May 20–24, 2013. IEEE, 716–725. DOI: https://doi.org/10.1109/IPDPSW.2013.8.

X. Défago, M. Gradinariu, S. Messika, and P. Raipin-Parvédy. 2006. Fault-tolerant and self-stabilizing mobile robots gathering. In *Proc. 20th Int. Symp. on Distributed Computing*. Springer, 46–60. DOI: https://doi.org/10.1007/11864219_4.

X. Défago, M. Potop-Butucaru, and S. Tixeuil. 2019. Fault-tolerant mobile robots. In P. Flocchini, G. Prencipe, and N. Santoro (Eds.), *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, Vol. 11340: Lecture Notes in Computer Science. Springer, 234–251. DOI: https://doi.org/10.1007/978-3-030-11072-7_10.

X. Défago, M. Potop-Butucaru, and P. R. Parvédy. 2020. Self-stabilizing gathering of mobile robots under crash or Byzantine faults. *Distrib. Comput.* 33, 5, 393–421. DOI: https://doi.org/10.1007/s00446-019-00359-x.

Y. Dieudonné and F. Petit. 2012. Self-stabilizing gathering with strong multiplicity detection. *Theor. Comput. Sci.* 428, 47–57. DOI: https://doi.org/10.1016/j.tcs.2011.12.010.

E. W. Dijkstra. 1962. Over de sequentialiteit van processbeschijvingen. EWD35.

E. W. Dijkstra. 1973a. Self-stabilization in spite of distributed control. EWD391.

E. W. Dijkstra. 1973b. Self-stabilization with four-state machines. EWD392.

E. W. Dijkstra. 1973c. Self-stabilization with three-state machines. EWD396.

E. W. Dijkstra. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11, 643–644. DOI: https://doi.org/10.1145/361179.361202.

E. W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8, 453–457. DOI: https://doi.org/10.1145/360933.360975.

E. W. Dijkstra. 1982a. Trip report E. W. Dijkstra Summer School Munich. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 31–33. Original date of EWD385 is August 7, 1973. DOI: https://doi.org/10.1007/978-1-4612-5695-3_4.

E. W. Dijkstra. 1982b. The solution to a cyclic relaxation problem. In *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 34–35. Original date of EWD386 is August 8, 1973. DOI: https://doi.org/10.1007/978-1-4612-5695-3_5.

E. W. Dijkstra. 1982c. *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag. DOI: https://doi.org/10.1007/978-1-4612-5695-3.

E. W. Dijkstra. 1986. A belated proof of self-stabilization. *Distrib. Comput.* 1, 1, 5–6. DOI: https://doi.org/10.1007/BF01843566.

E. W. Dijkstra. 1987a. Twenty-eight years. EWD1000.

E. W. Dijkstra. 1987b. Position paper on "fairness." EWD1013.

E. W. Dijkstra. 1989. In reply to comments. EWD1058.

E. W. Dijkstra and C. S. Scholten. 1979. Termination detection for diffusing computations. EWD687.

S. Dolev. 2000. *Self-Stabilization*. MIT Press.

S. Dolev and T. Herman. 1999. Parallel composition of stabilizing algorithms. In A. Arora (Ed.), *Proc. 19th IEEE Int. Conf. on Distributed Computing Systems*. IEEE, 25–32. DOI: https://doi.org/10.1109/SLFSTB.1999.777483.

S. Dolev and J. L. Welch. 2004. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM* 51, 5, 780–799. DOI: https://doi.org/10.1145/1017460.1017463.

S. Dolev, A. Israeli, and S. Moran. 1990. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. ACM SIGACT-SIGOPS 9th Symp. on the Principles of Distributed Computing*. ACM, 103–117. DOI: https://doi.org/10.1145/93385.93407.

S. Dolev, A. Israeli, and S. Moran. 1993. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distrib. Comput.* 7, 1, 3–16. DOI: https://doi.org/10.1007/BF02278851.

S. Dolev, M. Gouda, and M. Schneider. 1999. Memory requirements for silent stabilization. *Acta Inform.* 36, 6, 447–462. DOI: https://doi.org/10.1007/s002360050180.

S. Dolev, R. I. Kata, and E. M. Schiller. 2010. When consensus meets self-stabilization. *J. Comput. Syst. Sci.* 76, 884–900. DOI: https://doi.org/10.1016/j.jcss.2010.05.005.

S. Dubois and S. Tixeuil. 2011. A taxonomy of daemons in self-stabilization. *CoRR*, abs/1110.0334. https://arxiv.org/abs/1110.0334.

S. Dubois, T. Masuzawa, and S. Tixeuil. 2012. Bounding the impact of unbounded attacks in stabilization. *IEEE Trans. Parallel Distrib. Syst.* 23, 3, 460–466. DOI: https://doi.org/10.1109/TPDS.2011.158.

S. Dubois, T. Masuzawa, and S. Tixeuil. 2015. Maximum metric spanning tree made Byzantine tolerant. *Algorithmica* 73, 1, 166–201. DOI: https://doi.org/10.1007/s00453-014-9913-5.

B. Ducourthial and S. Tixeuil. 2001. Self-stabilization with r-operators. *Distrib. Comput.* 14, 3, 147–162. DOI: https://doi.org/10.1007/PL00008934.

N. Fallahi, B. Bonakdarpour, and S. Tixeuil. 2013. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013*, Braga, Portugal, October 1–3, 2013. IEEE Computer Society, 153–162. DOI: https://doi.org/10.1109/SRDS.2013.24.

M. Feldmann, C. Scheideler, and S. Schmid. 2020. Survey on algorithms for self-stabilizing overlay networks. *ACM Comput. Surv.* 53, 4, 74:1–74:24. DOI: https://doi.org/10.1145/3397190.

J. A. Fernández-Zepeda and J. P. Alvarado-Magaña. 2008. Analysis of the average execution time for a self-stabilizing leader election algorithm. *Int. J. Found. Comput. Sci.* 19, 6, 1387–1402. DOI: https://doi.org/10.1142/S0129054108006340.

P. Flocchini, G. Prencipe, and N. Santoro. 2008. Self-deployment of mobile sensors on a ring. *Theor. Comput. Sci.* 402, 1, 67–80. DOI: https://doi.org/10.1016/j.tcs.2008.03.006.

P. Flocchini, G. Prencipe, and N. Santoro. 2012. Distributed computing by oblivious mobile robots. *Synth. Lect. Distrib. Comput. Theory* 3, 2, 1–185. Morgan & Claypool. DOI: https://doi.org/10.2200/S00440ED1V01Y201208DCT010.

C. Genolini and S. Tixeuil. 2002. A lower bound on dynamic k-stabilization in asynchronous systems. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, 13–16 October 2002, Osaka, Japan. IEEE, 212–221. DOI: https://doi.org/10.1109/RELDIS.2002.1180190.

S. Ghosh. 2006. *Distributed Systems: An Algorithmic Approach*. Chapman and Hall/CRC.

S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. 2007. Fault-containing self-stabilizing distributed protocols. *Distrib. Comput.* 20, 1, 53–73. DOI: https://doi.org/10.1007/s00446-007-0032-2.

M. G. Gouda and N. Multari. 1991. Stabilizing communication protocols. *IEEE Trans. Comput.* 40, 4, 448–458. DOI: https://doi.org/10.1109/12.88464.

M. G. Gouda, J. A. Cobb, and C. Huang. 2006. Fault masking in tri-redundant systems. In A. K. Datta and M. Gradinariu (Eds.), *Proceedings of the Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006*, Dallas, TX, November 17–19, 2006, Vol. 4280: Lecture Notes in Computer Science. Springer, 304–313. DOI: https://doi.org/10.1007/978-3-540-49823-0_21.

M. Gradinariu and S. Tixeuil. 2000. Self-stabilizing vertex coloration and arbitrary graphs. In *Proceedings of the 4th International Conference on Principles of Distributed Systems*. Studia Informatica Universalis, Suger, Saint-Denis, rue Catulienne, France, 55–70. ISBN 2-912590-11-6.

T. Herman. 1990. Probabilistic self-stabilization. *Inf. Process. Lett.* 35, 2, 63–67. DOI: https://doi.org/10.1016/0020-0190(90)90107-9.

T. Herman. 1992. Self-stabilization: Randomness to reduce space. *Distrib. Comput.* 6, 2, 95–98. DOI: https://doi.org/10.1007/BF02252680.

T. Herman. 2003. Models of self-stabilization and sensor networks. In S. R. Das and S. K. Das (Eds.), *Proceedings of the Distributed Computing—IWDC 2003, 5th International Workshop*, Kolkata, India, December 27–30, 2003, Vol. 2918: Lecture Notes in Computer Science. Springer, 205–214. DOI: https://doi.org/10.1007/978-3-540-24604-6_20.

T. Herman and S. V. Pemmaraju. 2000. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.* 73, 1–2, 41–46. DOI: https://doi.org/10.1016/S0020-0190(99)00164-7.

T. Herman and T. Masuzawa. 2001. Available stabilizing heaps. *Inf. Process. Lett.* 77, 2–4, 115–121. DOI: https://doi.org/10.1016/S0020-0190(00)00208-8.

T. Herman and S. Tixeuil. 2004. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *Proceedings of the Algorithmic Aspects of Wireless Sensor Networks: First International Workshop, ALGOSENSORS 2004*, Turku, Finland, July 16, 2004, Vol. 3121: Lecture Notes in Computer Science. Springer, 45–58. DOI: https://doi.org/10.1007/978-3-540-27820-7_6.

S. Hsu and S. Huang. 1992. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.* 43, 2, 77–81. DOI: https://doi.org/10.1016/0020-0190(92)90015-N.

S. Katz and K. J. Perry. 1993. Self-stabilizing extensions for message-passing systems. *Distrib. Comput.* 7, 1, 17–26. DOI: https://doi.org/10.1007/BF02278852.

S. Köhler and V. Turau. 2012. Fault-containing self-stabilization in asynchronous systems with constant fault-gap. *Distrib Comput.* 25, 3, 207–224. DOI: https://doi.org/10.1007/s00446-011-0155-3.

H. S. M. Kruijer. 1979. Self-stabilization (in spite of distributed control) in tree-structured systems. *Inf. Process. Lett.* 8, 2, 91–95. DOI: https://doi.org/10.1016/0020-0190(79)90151-0.

A. D. Kshemkalyani and M. Singhal. 2011. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press.

S. Kutten and D. Peleg. 1995. Fault-local distributed mending (extended abstract). In J. H. Anderson (Ed.), *Proc. ACM SIGACT-SIGOPS 14th Symp. on the Principles of Distributed Computing*. ACM, 20–27. DOI: https://doi.org/10.1145/224964.224967.

L. Lamport. 1979. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.* 1, 1, 84–97. DOI: https://doi.org/10.1145/357062.357068.

L. Lamport. 1984. Solved problems, unsolved problems and non-problems in concurrency, invited address. In *Proc. ACM SIGACT-SIGOPS 3rd Symp. on the Principles of Distributed Computing*. ACM, 1–11. DOI: https://doi.org/10.1145/800222.806731.

L. Lamport. 1986a. On interprocess communication: Parts I and II. *Distrib. Comput.* 1, 2, 77–101. DOI: https://doi.org/10.1007/BF01786227 and https://doi.org/10.1007/BF01786228.

L. Lamport. 1986b. The mutual exclusion problem: Part II. Statement and solutions. *J. ACM* 33, 2, 327–348. DOI: https://doi.org/10.1145/5383.5385.

T. Li, J. A. Ambrose, R. Ragel, and S. Parameswaran. 2016. Processor design for soft errors: Challenges and state of the art. *ACM Comput. Surv.* 49, 3, 1–44. DOI: https://doi.org/10.1145/2996357.

T. Masuzawa and S. Tixeuil. 2006. A self-stabilizing link-coloring protocol resilient to unbounded Byzantine faults in arbitrary networks. In *Proc. 9th Int. Conf. Principles of Distributed Systems*, Vol. 3974: Lecture Notes in Computer Science. Springer, 118–129. DOI: https://doi.org/10.1007/11795490_11.

J. Misra. 1986. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.* 8, 1, 142–153. DOI: https://doi.org/10.1145/5001.5007.

M. Nesterenko and A. Arora. 2002. Tolerance to unbounded Byzantine faults. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, October 13–16, 2002, Osaka, Japan. IEEE Computer Society, 22–29. DOI: https://doi.org/10.1109/RELDIS.2002.1180170.

F. Ooshita and S. Tixeuil. 2015. On the self-stabilization of mobile oblivious robots in uniform rings. *Theor. Comput. Sci.* 568, 84–96. DOI: https://doi.org/10.1016/j.tcs.2014.12.008.

R. Stutsman, C. Lee, and J. Ousterhout. 2015. Experience with rules-based programming for distributed, concurrent, fault-tolerant code. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, 17–30.

Y. Sudo, A. K. Datta, L. L. Larmore, and T. Masuzawa. 2018. Constant-space self-stabilizing token distribution in trees. In Z. Lotker and B. Patt-Shamir (Eds.), *Structural Information and Communication Complexity—25th International Colloquium, SIROCCO 2018*, Ma'ale HaHamisha, Israel, June 18–21, 2018, Revised Selected Papers, Vol. 11085: Lecture Notes in Computer Science. Springer, 25–29. DOI: https://doi.org/10.1007/978-3-030-01325-7_4.

I. Suzuki and M. Yamashita. 1999. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.* 28, 4, 1347–1363. DOI: https://doi.org/10.1137/S009753979628292X.

M. Tchuente. 1981. Sur l'auto- stabilisation dans un réseau d'ordinateurs. *RAIRO - Theor. Inform. Appl.* 15, 1, 47–66. http://www.numdam.org/item?id=ITA_1981__15_1_47_0.

G. Tel. 2000. *Introduction to Distributed Algorithms* (2nd. ed.). Cambridge University Press.

S. Tixeuil. 2009. Self-stabilizing algorithms. In M. J. Atallah and M. Blanton (Eds.), *Algorithms and Theory of Computation Handbook* (2nd. ed.). CRC Press, Taylor & Francis Group, 26.1–26.45. ISBN 978-1-58488-820-8.

G. Varghese. 2000. Self-stabilization by counter flushing. *SIAM J. Comput.* 30, 2, 486–510. DOI: https://doi.org/10.1137/S009753979732760X.

G. Varghese and M. Jayaram. 2000. The fault span of crash failures. *J. ACM* 47, 2, 244–293. DOI: https://doi.org/10.1145/333979.333982.

W. Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1, 40–44. DOI: https://doi.org/10.1145/1435417.1435432.

6

Dijkstra's Legacy on Program Verification

Reiner Hähnle

> [The Computing Scientist] is familiar and totally at ease with the idea of manipulating uninterpreted formulae. [...] [H]e has good reason for not sharing the common fear of symbol manipulation because he has the tools for mechanization at his disposal.
> —Edsger W. Dijkstra [1988]

The decade from 1967 to 1977 was one of the most creative in the field of program verification, with the advent of proof assertions, program logics, weakest preconditions, logic programming, symbolic execution, denotational and relational semantics, logical frameworks, the first program verifiers, to name just a few achievements. Dijkstra was at the center of these developments and several seminal ideas are due to him.[1] In this article I try to position his contributions to program verification in their historical but also conceptual context, and I trace the legacy of Dijkstra's ideas on program verification.

Before we begin, we need to make a clarification: the term *program verification*, as used by Dijkstra, refers to the activity of proving the correctness of a program with respect to a detailed specification for *all possible runs*. He sharply (and famously) contrasted verification to testing: "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra n.d.].[2] Nowadays, the term *verification* is used in a much broader sense in Software Engineering and

[1] The involvement of Dijkstra in early efforts on program verification is described in detail in Chapter 7.

[2] See also EWD 303 reproduced in this book.

includes all kind of activities that increase trust in a program's correctness, including testing and various static analyses. Here we use the term program verification in Dijkstra's sense.

6.1 Dijkstra's View of Program Verification

Dijkstra viewed program verification as an activity not to be separated from specification and design. He also viewed programming and, in particular, reasoning about the correctness of programs essentially as a mathematical activity.

6.1.1 A Posteriori Verification versus Correctness-by-Construction

We start with some terminology: *a posteriori verification* (also called *post hoc verification* or phv for short) designates the approach where a program is verified *after* it has been constructed and, possibly, even deployed. Dijkstra was critical of *a posteriori* verification and promoted a *constructive* approach, often termed *correctness-by-construction*, cbc for short. The starting point of the latter is not the program code but a mathematical formalization of a program's intended behavior (a specification), from which in a series of precise steps a correctness proof together with executable code is gradually developed. Dijkstra [1976b] motivates the problem thus:

> [. . .] for different programs meeting the same specifications, the corresponding correctness proofs could greatly differ in complexity! [. . .] the program should be designed in such a way that its correctness could, indeed, be established. The simplicity of the corresponding correctness proof became thus an important aspect of program quality. [. . .] [H]ow does one develop a program that admits a nice correctness proof? Well, by developing the program and its correctness proof hand in hand.

In his seminal book *A Discipline of Programming* [Dijkstra 1976a], Dijkstra proposed the weakest precondition calculus that was designed to support cbc. In a historical account of computer science from the 1950s onwards, Dijkstra [1988] contrasts phv with cbc and locates both (for the 1970s decade) within different scientific cultural contexts:

> [T]he American interest focused on the mechanization of a posteriori verification of given programs, written in given languages; the European interest was more in a constructive approach to the problem of program correctness, and turned to design methodologies or calculi for the derivation of programs that would be correct by construction. From the European perspective, a posteriori program verification amounted to putting the cart before the horse;

> from the American perspective, the constructive approach of the Europeans was hopelessly idealistic, because it was mathematical.

In the same article he (tentatively) characterized the subsequent 1980s as a "decade of synthesis" between Computer Science[3] and formal mathematical concepts. I believe this judgment was perspicacious. In fact, the 1980s initiated an *era* of synthesis, not merely a decade, that is still unfolding, see Section 6.4.2 below.

Behind Dijkstra's metaphor of "putting the cart before the horse" is his deep conviction, which he strove to realize during his whole professional life, that programming should be a *scientific* discipline, that scientific thought is not a luxury but a *necessity* in order to produce correctly working software [Dijkstra 1982b]. It is hard enough to understand and to formulate the mathematical properties underlying a complex algorithm, but when this task is performed not systematically, and for programs written in an idiosyncratic language, then arguing for correctness turns into a hopelessly complex endeavor. To ensure correctness of programs that solve difficult problems, he deems two stratagems to be of essence:

(I) Strip away unnecessary features from the target programming language while keeping it universal: indeed, his guarded command language [Dijkstra 1976a] (see Section 6.2.3) does not feature complex types, jumps, or even procedures.

(II) Achieve a separation of concerns for the tasks at hand: For example, it is simply too hard to simultaneously understand, code, and prove correctness of a complex algorithm. To disentangle these tasks, first specify the requirements, then—step by step—develop the specification into a provably correct, executable program. This is the approach proposed in Dijkstra [1976a].

Both points serve to *reduce complexity* in order to make the task of verification manageable. Dijkstra [1989a] stresses:

> Computing's core challenge is how not to make a mess of it. [. . .] all unmastered complexity is of our own making; there is no one else to blame and so we had better learn how not to introduce the complexity in the first place.

6.1.2 Criticism

One main objection to the "constructive" approach can be illustrated with a quote attributed to Bjarne Stroustrup, the designer of the C++ language: "There are only two kinds of languages: the ones people complain about and the ones nobody

[3] Dijkstra preferred the term Computing Science, but we use the more established term.

uses."[4] From today's state of program verification [Hähnle and Huisman 2019] one can observe a partial reconciliation between usability and analyzability of programming languages: one important development is that high-level features, such as strong type systems or object encapsulation made it into mainstream programming languages. Problematic constructs, such as **go to**, were excised. Program verification and requirements specification is done in high-level languages, while low-level languages are supported via compilation. To this end, several large projects endeavored to produce verified compilers [Fournet et al. 2016, Heiser et al. 2020].

One must also mention methodological advances that considerably improved modularity in program verification in the presence of advanced language features, such as procedures, reference types, or objects [Gurov et al. 2020]. This is an essential step towards complexity reduction. Dijkstra might have approved of these developments. He was not a fundamentalist who admitted only minimalist notation. As highlighted by the quote above, what he fought against was to introduce complexity not inherently justified by the problem at hand.

Another objection raised against Dijkstra's views is that most programmers (and this was even more true in Dijkstra's time) lack the mathematical training (or even a Computer Science background) required to practice cbc. Dijkstra retorted that "past experience has made me a firm believer that [. . .] how to avoid unmastered complexity, can indeed be taught" [Dijkstra 1982c], ranted against "complexity generators" willfully produced by unscientific approaches, and the unsatisfactory state of training in Computing in general [Dijkstra 1995].[5] One can observe that there are many more people around today with formal university degrees in Computer Science than 25 years ago and most curricula at major institutions contain training in formal aspects. On the other hand, different kinds of adverse forces have grown considerably, most notably perhaps the remorseless pressure to minimize "time-to-market" [Bosch 2016].

My own misgivings concerning the constructive approach have a different flavor and can—tongue in cheek—be illustrated as follows: in old-fashioned Western movies one can sometimes see the hero ("the specification") standing on the roof of a moving train wagon. He aims with his gun at the villain ("the brilliant algorithm"), who is riding on a galloping horse next to the train and shoots him right

[4] Stroustrup writes that "two" has to be taken with a pinch of salt and doubts having been the first person to phrase the quoted sentiment, see https://www.stroustrup.com/quotes.html.

[5] "[. . .] in the near future we shall have to live with the superstition that also programming is 'so easy that even a Republican can do it'." The current AI hype brought this "superstition" more to the forefront than ever—Dijkstra would not have been pleased.

between the eyes. In more sober terms: it is not clear how the *discipline* would help someone with a less good aim than Dijkstra to derive a beautiful algorithm from a declarative specification. Time and again, in the case studies of Dijkstra [1976a] algorithmic ideas pop up from seemingly nowhere. Without doubt it is easier to prove as correct a program that was designed for verification. But I cannot completely shed the impression that many times the cbc development process is a reconstruction of how the development ideally *should* have been: the bullet is carefully guided by an invisible hand.

6.1.3 Informal, Rigorous, Formal, Mechanized, Automated

The style of *A Discipline of Programming* [Dijkstra 1976a] is mathematically rigorous, but not formal. For example, specifications are written as formulas, but the syntax and semantics of the specification language is not formally defined. This was caused by the fact that by the mid-1970s Dijkstra was still struggling to find appropriate concepts for his endeavor, formalization was not the main concern.[6] Still, the presentation is sufficiently rigorous so it could later be formalized in *Predicate Calculus and Program Semantics* [Dijkstra and Scholten 1990] without too many modifications.[7] In the following, I will refer to the older book with the acronym *DoP* and to the more recent one with *PCPS*.

Notational rigor, precision, and conciseness were of utmost importance to Dijkstra. He frequently criticized "traditional" mathematicians for their lack of rigor, for introducing unnecessary complexity, and for using unintuitive notation. He strongly objected to the "consensus model" in mathematics ("a proof was deemed correct [. . .] when it was accepted by the mathematical colleagues because they saw nothing wrong [. . .] with it" [Dijkstra 1988]) and became enraged when DeMillo et al. [1977] attempted to use this argument to show that program verification is a futile enterprise ("a very ugly paper," "the style of a political pamphlet," "the paper is pre-scientific" [Dijkstra 1977a]). Dijkstra favored "calculational arguments" based on "syntactically unambiguous" "formula manipulation." He argued that this style of mathematical argument is natural to computer scientists:[8]

> Computing scientists, as a rule, tend to be aware of it; familiar as they are with manipulation (mechanized or not) of uninterpreted formulae, they tend to feel quite at home with more calculational arguments. Moreover they

[6] At the end of Dijkstra [1976a], in the chapter "In Retrospect," the genealogy of his key design decisions is explained.

[7] Dijkstra's approach to formalizing correctness proofs is discussed in Chapter 11 written by Vladimir Lifschitz.

[8] The previous and subsequent quotations are all from Dijkstra [1988].

> need the formal arguments as they cope better than the verbal arguments with the scaling up that reasoning about digital systems quickly requires.

Clearly, even if mechanization and automation of program verification was not in the focus of Dijkstra's research interests, his whole research agenda on program verification was shaped by the possibility of admitting mechanical, tool-assisted proofs.

6.2 The Weakest Precondition Calculus and Its Context

Arguably, Dijkstra's most important and lasting contribution to the field of program verification is his *weakest precondition calculus*, wp-calculus for short, which is at the center of DoP.

6.2.1 Setting the Stage

To put the contributions in DoP into context, one must recall the state of art in programming language semantics in the early 1970s. The dominating approach was *operational semantics* [McCarthy 1966], disliked by both Dijkstra and Hoare.[9] An operational semantics for a programming language essentially defines an interpreter by specifying for each abstract syntax element the effect its execution has on the program state. Rather than just specifying how instructions are executed, Dijkstra (and earlier Hoare) preferred a semantics that would explain the underlying language concepts at a more abstract level.

Neither did Dijkstra see any use for relational semantics proposed by De Bakker and De Roever [1972]. A comparative account of various semantics of the wp-calculus by Plotkin [1979] is not mentioned anywhere in his notes or in PCPS. In general, he was highly skeptical of what he deemed overly theoretical approaches. The following seething quotation seems to confirm that he dismissed relational semantics on this ground [Dijkstra 1974a]:

> "[T]heoretical computing science" [. . .], I am afraid, is just a euphemism for "unpractical computing science"; and the latter could very well turn out to be a *contradictio in terminis*. [. . .] the new journal (!) "Theoretical Computer Science" is no more than the next political move of the French mathematicians, who seem to fear but one thing: to lose their power. I guess that they will succeed in preventing the birth of Computing Science in France for at least another decade [. . .]. I am willing to make a bet that de Bakker will publish in it . . . [10]

[9] Hoare recalls in Chapter 25 that "I shared his distaste for operational semantics".

[10] The *Theoretical Computer Science* journal became highly successful and for many years was the largest journal in Computer Science.

6.2.2 Floyd and Hoare

Floyd [1967] made a first step towards an *axiomatic semantics* by attaching assertions to *flowchart programs* that had to respect certain axioms derived from the intended program semantics. In 1969, Hoare published his seminal article [Hoare 1969], where he introduced an axiomatic, rule-based approach to programming language semantics. The target language formed *while-programs* from side effect-free arithmetic and Boolean expressions, program variables over mathematical integers, assignments, **if**–**then** conditionals, and **while**–**do** loops.[11]

The central concept of Hoare's formalism is what today is known as a *Hoare triple*: If S is a program segment and P, Q are first-order assertions over variables in S then the expression[12]

$$\{P\}\,S\,\{Q\} \tag{6.1}$$

means: if S is started in a state, where P holds and it terminates, then in its final state Q holds. If P does not hold initially, or if S does not terminate, then (6.1) holds trivially. In the current terminology, a Hoare triple expresses *partial correctness* of S relative to precondition P and postcondition Q.

The semantics of a programming language can now be expressed with the help of schematic axioms and inference rules over Hoare triples. For example, the axiom schema for assignment is

$$\{(x := E).P\}\,x := E\,\{P\}, \tag{6.2}$$

where P is an assertion, x a program variable over a scalar data type, E a side-effect free expression of the same type as x, and $(x := E).P$ means that any free occurrence of x in P is replaced with E.[13]

The inference rule for sequential composition is given as:

$$\frac{\{P\}\,S_0\,\{R\} \qquad \{R\}\,S_1\,\{Q\}}{\{P\}\,S_0;\ S_1\,\{Q\}}, \tag{6.3}$$

where S_0, S_1 are program segments and P, Q, R assertions. The rule has to be read as follows: if we can prove $\{P\}\,S_0\,\{R\}$ and $\{R\}\,S_1\,\{Q\}$, then we have also proven $\{P\}\,S_0;\ S_1\,\{Q\}$.

[11] Hoare followed up his foundational work with an axiomatic semantics for several complex language constructs, including almost the full Pascal language [Hoare and Wirth 1973].

[12] In his first papers, Hoare had put the curly braces around the program, not the assertions. Soon after Apt [1981], the convention was reversed. The latter is convenient because additional assertions can be freely inserted in the program text by putting braces around them.

[13] Using the notation of PCPS [Dijkstra and Scholten 1990] for substitutions.

Rule (6.3) illustrates that the axiomatic approach offers a higher level of abstraction than operational semantics: it stipulates that there is *some* intermediate state between S_0 and S_1, but not how this state is reached. Moreover, there is a clear separation of concern between programs and requirements.

Perhaps the most significant consequence of axiomatic program semantics is that, unlike relational semantics, it renders itself directly as a proof method to reason about program correctness. This was recognized by both Floyd [1967] and Hoare [1969], whose papers feature examples of correctness proofs. Already King [1969] prototyped a verification tool based on Floyd's method.

The main drawback of Hoare's approach is that it is incomplete as a programming language semantics because the characterization of loops is based on the invariant rule:

$$\frac{\{I \wedge B\}\; S\; \{I\}}{\{I\}\;\; \mathbf{while}\; B\; \mathbf{do}\; S\; \{I \wedge \neg B\}}. \tag{6.4}$$

Any assertion I that stays invariant under execution of the loop body (assuming B holds in the beginning) also holds when the loop terminates (in which case $\neg B$ holds). Invariants provide a sufficient criterion to be used in correctness proofs, but they fail to *characterize* the semantics of loop statements. In fact, it is not even possible in all cases to formulate sufficiently strong invariants to establish a desired postcondition [Blass and Gurevich 2001]. In contrast, the wp-calculus in DoP aimed at a programming language semantics that fully characterizes loops.

6.2.3 Guarded Commands

Dijkstra developed his wp-calculus hand in hand with the *guarded command* programming language. Conceptually, the calculus can stand on its own, as we shall see below, but we stick to guarded commands in our presentation because this is what Dijkstra intended.

The guarded command language is the focus of DoP and PCPS but differs between both books. In DoP, guarded command programs may declare local and global, private and public variables, as well as constants with their scope declarations, but without a formal semantics: The case studies abstract away from scoping and visibility rules. DoP also features scalar arrays with dynamic bounds. All these features are left out from the more formal account in PCPS that we take as the basis of our discussion. In neither version does the guarded command language have procedure calls or statements for concurrency. Obviously, given Dijkstra's well-known disapproval of the **go to** statement,[14] there are no jump statements either.

[14] Dijkstra [1968], reproduced in the present book.

The guarded command language is so compact that we can give the gist of it in a single page, but its ramifications are vast. A more detailed account of guarded commands and the nondeterminism entailed by them is given in Chapter 8 written by Krzysztof R. Apt and Ernst-Rüdiger Olderog.

The syntax of expressions, assignment statements (generalized to simultaneous assignments), and sequential composition is like in Hoare logic. There is a *skip* statement with obvious semantics and an *abort* statement, from which no satisfiable final state can ever be reached.

Conditionals **if** B **then** S are generalized to an *alternative statement* of the form

$$\mathbf{if}\ B_0 \longrightarrow S_0 [\!] \cdots [\!] B_{n-1} \longrightarrow S_{n-1}\ \mathbf{fi},$$

where B_i are Boolean expressions and S_i are statements. Each clause $B_i \longrightarrow S_i$ is called a *guarded command* with *guard* B_i. The empty list of guarded commands ($n = 0$) is permitted.

The semantics of the alternative statement is that one *arbitrary* statement, whose guard is true in the current state, is executed. This has the immediate consequence that programs become nondeterministic. For example, the program **if** $\mathit{true} \longrightarrow x := 0 [\!] \mathit{true} \longrightarrow x := 1$ **fi** might result in a state in which x is either 0 or 1. If none of the guards is true, then execution aborts. For $n = 0$ we have **if fi** ≡ *abort*. The alternative statement with a single guarded command ($n = 1$) is *not* the same as a conditional: the latter, in case the guard is false, does not abort but resumes with the subsequent statement.

In analogy to conditionals, loops **while** B **do** S are generalized to a *repetition statement* of the form

$$\mathbf{do}\ B_0 \longrightarrow S_0 [\!] \cdots [\!] B_{n-1} \longrightarrow S_{n-1}\ \mathbf{od}$$

with the semantics that as long as one of the guards is true in the current state, the corresponding command is nondeterministically selected and the proviso that the repetition statement terminates once none of the guards is true. For $n = 1$, the repetition statement degenerates into a while loop and is deterministic provided that the code in its body is. For $n = 0$, we have **do od** ≡ *skip*.

Many algorithms can be elegantly formulated with the repetition statement that combines iteration with choice. For example, this program implementing Euclid's greatest common divisor (gcd) algorithm for two positive numbers X, Y

```
$x, y := X, Y;$
  do $x > y \longrightarrow x := x - y$
     [] $y > x \longrightarrow y := y - x$
  od /* $\mathrm{gcd}(X, Y) = x = y$ */
```

avoids a conditional inside the loop body and treats both variables in a symmetric way.

6.2.4 The Weakest Precondition Calculus

The central concept of Dijkstra's wp-calculus is what he called a *predicate transformer*:[15]

$$\mathrm{wp}.S.Q, \tag{6.5}$$

where S is a program segment and Q is a first-order formula.[16] More precisely, wp is a function on programs S such that wp.S returns a mapping from formulas to formulas. Expression (6.5) represents the logically weakest formula P such that, if S is started in a state, in which P holds, then it terminates and in its final state Q holds. In other words, P is the[17] *weakest precondition* for S that necessarily establishes *postcondition* Q and ensures termination of S. If S does not terminate, then wp.$S.Q$ is false.[18]

How Dijkstra arrived at his weakest precondition calculus is traced in Chapter 7, written by David Gries, who also discusses Dijkstra's original motivation: to develop programs together with their formal proofs. Here we focus on the relevance of the weakest preconditions for mechanical verification. First, we explain their relation to Hoare's approach.

If we represent while loops "**while** B **do** S" as guarded commands "**do** $B \longrightarrow S$ **od**", then for while-programs the relation between predicate transformers and Hoare triples[19] is obvious: assuming S terminates, whenever Hoare triple (6.1) holds then $P \implies \mathrm{wp}.S.Q$. In particular, $\{\mathrm{wp}.S.Q\}\, S\, \{Q\}$ holds. The implication from (6.1) to $P \implies \mathrm{wp}.S.Q$ only holds for terminating programs: for a nonterminating program S and postcondition Q, the Hoare triple $\{\mathit{true}\}\, S\, \{Q\}$ holds trivially, but wp.$S.Q$ is false. On the other hand, $\{\mathrm{wp}.S.Q\}\, S\, \{Q\}$ is always valid.

Dijkstra defined a second predicate transformer, the *weakest liberal precondition* wlp.$S.Q$ corresponding to partial correctness. In this case, the Hoare triple (6.1) always implies $P \implies \mathrm{wlp}.S.Q$.

In addition to the guarded command language syntax, there are three main differences between Hoare's and Dijkstra's setup: (i) the treatment of termination,

[15] We mostly follow (with a few liberties) the notation of Dijkstra and Scholten [1990].

[16] DoP uses the terms "predicate" and "condition" instead of formula, hence *predicate* transformer. PCPS employs the rather unfortunate terminology *(boolean) structure*.

[17] Obviously, P is unique only up to logical equivalence.

[18] In DoP, wp.S is a partial function that is undefined in case of non-termination.

[19] In their original form for partial program correctness.

(ii) guarded command programs are nondeterministic, and (iii) loop statements are given a semantic characterization.

The schematic transformation rules defining the predicate transformer for statements S constitute the axiomatic[20] semantics of guarded command programs. To understand the challenges posed by the use of the weakest precondition calculus in mechanical verification, we need to review the definitions. We begin with single assignments and sequential composition:

$$\mathrm{wp.}\,(x := E)\,.Q = (x := E).Q \tag{6.6}$$

$$\mathrm{wp.}\,(S_1;\; S_2)\,.Q = \mathrm{wp.}S_1.(\mathrm{wp.}S_2.Q). \tag{6.7}$$

Rule (6.6) corresponds directly to axiom (6.2), whereas rule (6.7) is more specific than rule (6.3) because it suggests to analyze a program "backwards," in the opposite direction of the control flow: starting with the postcondition Q. Then for the final statement S_{fin} in a sequential composition, $\mathrm{wp.}S_{\text{fin}}.Q$ is used in turn as the postcondition of the second-but-last statement, and so on. In contrast, rule (6.3) allows one to deal with a statement sequence in arbitrary order.[21]

Of course, there are good reasons for the setup Dijkstra chose: first, in languages with declaration blocks or termination caused by exceptions or breaks, rule (6.3) is not applicable. What about the *forward* style reasoning then? PCPS [Dijkstra and Scholten 1990, p. 215] states two reasons against it: first, for a given precondition P and statement $x := E$, it is necessary to compute the *strongest postcondition* in that case, that is, the logically strongest formula that holds in the post-state, when $x := E$ is executed in a state satisfying P. The strongest postcondition of an assignment is more awkward to compute than its weakest precondition because one needs to record the initial value before the assignment in a fresh variable. Second, strongest postconditions do not address partial correctness. While both arguments indeed favor weakest precondition semantics, they do not necessarily enforce backward style program verification. We return to this issue in Section 6.3.4.

Next, we look at the wp-characterization of the alternative statement:

$$\mathrm{wp.}\left(\mathbf{if}\,B_0 \longrightarrow S_0 [\!] \cdots [\!] B_{n-1} \longrightarrow S_{n-1}\mathbf{fi}\right).\,Q = \left(\bigvee_{i=0}^{n-1} B_i\right) \wedge \left(\bigwedge_{i=0}^{n-1} (B_i \Rightarrow \mathrm{wp.}S_i.Q)\right). \tag{6.8}$$

[20] Following a consideration by Bertrand Russell pointed out by Hoare, Dijkstra preferred the term *postulational* over axiomatic semantics [Dijkstra 1974b]. Since this did not catch on in the literature, we stick to the latter.

[21] Because axiom (6.2) computes the precondition, however, the direction of proof when dealing with assignments tends to be backwards in Hoare logic too.

The first conjunct on the right ensures that a satisfiable precondition can be obtained only if one of the guards is true. The second conjunct computes the wp of each guarded command assuming its guard. This concludes the wp-characterization of "straight-line programs," that is, programs without repetition.

The wp-characterization of repetition statements is necessarily more complex because termination is not even semidecidable. In DoP, the semantics of repetitions is defined via unbounded recursion on predicate transformers, while in PCPS the semantics of a repetition statement wp. $\left(\mathbf{do}\, B_0 \longrightarrow S_0 [] \cdots [] B_{n-1} \longrightarrow S_{n-1}\mathbf{od}\right)$. Q is defined as the logically strongest[22] solution Y to the equation:

$$Y \equiv \left(\left(\bigvee_{i=0}^{n-1} B_i\right) \vee Q\right) \wedge \left(\bigwedge_{i=0}^{n-1} (B_i \Rightarrow \text{wp}.S_i.Y)\right). \tag{6.9}$$

In case of nontermination the strongest solution is *false*. Otherwise, Y is the logically strongest formula (not necessarily first-order) that (i) ensures termination of the loop, (ii) is invariant under all guarded command statements, and, (iii) conjoined with the negated guards, establishes postcondition Q. In particular, for the case $n = 1$ and programs that terminate when started in P, any *loop invariant* I in the sense of rule (6.4) such that $I \wedge \neg B \Rightarrow Q$ implies the solution to equation (6.9). Dijkstra records this fact (and its generalization to arbitrary n) as the *basic theorem* for repetition in DoP, the *main repetition theorem* in PCPS, respectively, and credits this result, albeit expressed informally, to Floyd. Its significance is that, for the purpose of program verification, it is unnecessary to precisely compute a loop's weakest precondition. Instead of using the wp-calculus to reason about a program, it suffices to come up with a sufficiently strong invariant. For example, for the gcd program at the end of Section 6.2.3, a sufficient invariant is $x \geq 0 \wedge y \geq 0 \wedge \text{gcd}(X, Y) = \text{gcd}(x, y)$.

Conversely, as Clarke [1979] observed, the weakest (liberal) precondition of a loop is always an invariant of that loop, a fact that is useful to simplify completeness proofs. Clarke also connected fixpoint computation with soundness and completeness in this way.

To summarize, when the wp-calculus is used for program verification, and not as a programming language semantics, one tries to find *sufficient* preconditions, not necessarily *weakest* ones. This is not merely of theoretical but of eminently practical value, not the least for Dijkstra's goal in DoP: all nontrivial examples and the case studies focus on the derivation of suitable invariants. So, Dijkstra *worked with two different formalisms*: the fixpoint construction in the wp-calculus is used to define a programming language semantics for guarded command programs,

[22] The solution can be obtained with the usual fixpoint construction, but this is not of interest here.

while invariants that *approximate* weakest preconditions are used for the purpose of program derivation and verification.

The reader probably noticed that I was careful to add the qualification "for terminating programs" in the discussion above whenever I compare the *wp-calculus* with *Hoare logic*. The decision to base wp on *total correctness* leads to more complex definitions. Dijkstra was aware of this issue and proposed a separation of concerns between proving partial correctness and termination in chapter 6 of DoP, justified by the equivalence wp.$S.Q \equiv$ wlp.$S.Q \wedge$ wp.$S.true$. To establish wp.$S.true$ for a loop, he suggested not to attempt to find the weakest precondition for its termination, which might be hard to formulate, but to use instead a *decreasing* (chapter 6, DoP) or *variant function* (chapter 9, PCPS). This is a nonnegative integer function guaranteed to decrease in each loop iteration. For example, to prove termination of the gcd program one can simply use $x + y$. That technique is routinely used to ensure loop termination in deductive verification tools (for example, Ahrendt et al. [2016, p. 334]).

The relation between loop invariants and weakest preconditions sketched above highlights the pivotal role invariants play in program verification [Furia et al. 2014]. Obviously, the big question is how to find suitable, that is, sufficiently strong invariants. This is generally not even possible in first-order logic, so a large number of heuristic approaches to generate loop invariants automatically have been proposed since the early days of program verification [Wegbreit 1973]. They all attempt to summarize computations performed in a loop body with various techniques, including templates, symbol elimination, abstract interpretation, computer algebra, learning, to name just a few. None of them is robust enough to work reliably outside a specific application domain and only a few of the ingenious loop invariants discovered by Dijkstra in DoP could have been found automatically.

An issue ignored by Dijkstra and his contemporaries, because they looked at one small program at a time, is how to avoid bloating of loop invariants. An invariant has to imply (together with the negated guard) the postcondition. But for larger programs the postcondition can be a huge expression, only a small fragment of which pertains to the loop body at hand. In consequence, for practical program verification, one needs techniques that distill from pre- or postconditions those parts that actually can be affected by a given loop body [Beckert et al. 2005].

6.2.5 Symbolic Execution

The program analysis technique *symbolic execution* was proposed independently by various authors [Deutsch 1973, Burstall 1974, Boyer et al. 1975, King 1976]. The main idea is very intuitive: a given program is interpreted with *symbolic* input values, giving rise to symbolic program states. As a consequence, whenever a conditional or a loop statement is executed, one cannot decide which branch in the control flow graph is taken next. Instead, *all* program paths in the current symbolic state need to be explored. The guard of such a feasible branch is added to the symbolic state as a *path condition*. For example, start the gcd program from Section 6.2.3 under the condition that X, Y are positive:

Next statement	Program state	Path condition[23]
$x, y{:=}X, Y;$	*ture*	$X > 0 \wedge Y > 0$

After the parallel assignment, we obtain:

do ⋯ **od**	$x = X \wedge y = Y$	$X > 0 \wedge Y > 0$

Since all loop guards are consistent with the program state and path condition, there are three subsequent symbolic states:

do ⋯ **od**	$x = X - Y \wedge y = Y$	$X > Y \wedge X > 0 \wedge Y > 0$
do ⋯ **od**	$x = X \wedge y = Y - X$	$Y > X \wedge X > 0 \wedge Y > 0$
⟨⟨end⟩⟩	$x = X \wedge y = Y$	$X = Y \wedge X > 0 \wedge Y > 0$

The third path reaches the end of the program (in a state where $x = y$, which is implied by the path condition), but from the first two states the loop can be unwound again. The obvious question is: how to handle unbounded loops such as in this example?[24] Obviously, just as in Hoare logic and in the wp-calculus, one can use loop invariants to approximate (and in some cases characterize) an unbounded number of executions of the loop body. Historically, a program verification tool based on this idea was already described by Deutsch [1973]. Burstall [1974] suggested a variation where induction over a program expression is used: the induction claim states that for all values of a variable x a state Q is reached after executing a loop. After unwinding the loop, the value of x decreased with respect to the induction order and the induction hypothesis can be applied.

Of course, one can perform *incomplete* symbolic execution, for example, until a given path depth. This can be useful for generating test cases by instantiating path conditions. In the last decade this became a mainstream technology [Baldoni et al. 2018]. Debugging and testing with symbolic execution was already suggested by Boyer et al. [1975] and King [1976].

[23] Path conditions refer only to the initial values X, Y.

[24] The precondition $X > 0 \wedge Y > 0$ does not help because the loop bound is symbolic and thus unknown.

Dijkstra [1982d] correctly identified the bottleneck[25] of symbolic execution: the combinatorial explosion of the number of program paths, even for modest depth. Only relatively recently, effective techniques to mitigate this problem were discovered [Krishnamoorthy et al. 2010].

The relation between symbolic execution and the wp-calculus is straightforward: symbolic execution can be viewed as computing the strongest postcondition of a straight-line program, that is, a program without repetition. Just as weakest preconditions are approximated by an invariant when encountering loops, the same happens when symbolic execution is used for program verification: in general, only a formula that is implied by the strongest postcondition can be computed.

To execute a program in the forward direction of control flow is natural when trying to understand its workings. The forward direction is also compatible with well-established informal quality assurance techniques, such as *code inspection* [Fagan 1976]. The advantage of proving correctness of a program in the forward direction for the user of an interactive program verifier was stressed by designers of early verification tools [Deutsch 1973, Galand and Loncour 1975], but this line was not continued because the *path explosion* issue loomed too large, while alternative architectures for program verifiers proved to be successful (Section 6.3.3) in the meantime. Techniques to address path explosion when symbolic execution is used for program verification are discussed in Section 6.3.4.

6.2.6 Relational Calculus

An assertion P over the variables in a program can be identified with the set of program states that satisfy P, that is, a unary test predicate over states. Similarly, a program can be seen as a binary relation on states: all pairs of states (s, t) such that the program terminates in t when started in s. In the *relational calculus* [De Bakker and De Roever 1972] the view is reversed and relations are seen as programs that map a given state to a possibly empty set of successor states. In this view, following Harel and Pratt [1978], a unary relation P is a primitive program that maps a state s to the singleton $\{s\}$ when P is true in s and to the empty set otherwise. Other ways to compose relations are union (nondeterministic choice) "$S_0 \cup S_1$", join (sequential composition) "$S_0; S_1$", and finite repetition "S^*".

A conditional "**if** B **then** S" is modeled in relational calculus as "$(B;\ S) \cup \neg B$", a while-loop "**while** B **do** S" as "$(B;\ S)^*; \neg B$". De Bakker in his book [Bakker et al. 1980], unaware of Dijkstra's critical remarks about the relational calculus, provided a

[25] This is the only mention of the term *symbolic execution* in the EWD notes. There is no evidence that Dijkstra was aware of Burstall's and Deutsch's results, even though he met Burstall on several occasions.

simple modeling of guarded commands within relational calculus. For example, "**if** $B_0 \longrightarrow S_0 [] \cdots [] B_{n-1} \longrightarrow S_{n-1}$**fi**" is modeled as "$\bigcup_{i=0}^{n-1}(B_i;\ S_i)$".

The advantage of relational calculus is that programs have an immediate semantics in terms of set-theoretic operations on states, which does not require sophisticated mathematical concepts. For example, termination needs no special treatment: if a program S diverges in state s, then the *relation* S simply does not contain any pair of the form (s, t). Also partial correctness relative to P and Q can be expressed simply as $P; S \subseteq Q$. On the other hand, since the relational calculus is based on set theory, unlike Hoare logic and the wp-calculus, it does not render itself directly as a mechanical proof system and was not used as such. Even performing correctness proofs by hand is tedious, but this was not the intention: to provide a mathematical semantics of programs based on set theory.

6.2.7 Dynamic Logic

The semantic formalisms discussed so far cannot be viewed as a *logic* in the traditional sense: in Hoare logic programs are not subformulas of Hoare triples, so the syntax is not closed. For this reason, in order to model complex programming language features and correctness properties, a myriad of *ad hoc* extensions of Hoare triples have been suggested. For example, the following papers all suggest some version of Hoare quadruple, each with a different syntax, semantics, and purpose: Yang [2007], Kojima and Igarashi [2013], Bubel and Hähnle [2016], and Rauch et al. [2017]. Neither the wp-calculus nor symbolic execution nor relational calculus are cast as a program *logic*, of which one could reasonably expect three things:

(I) a formula syntax including programs, ideally being closed with respect to first-order connectives and quantifiers,

(II) a calculus defining which formulas can be derived as theorems, and

(III) a semantics, defining truth and validity of formulas, as well as the behavior of programs, such that only valid theorems can be derived.

Such a logic was indeed proposed, around the time Dijkstra wrote up his ideas about weakest preconditions, by Salwicki [1970] under the term *algorithmic logic* and later, but independently, by Pratt [1976] under the term *dynamic logic*.[26] Item 3 above was the central motivation to develop dynamic logic, as confirmed by Pratt [2017]:

[26] As happened so many times, also Salwicki's work suffered from a lack of exposure and belated recognition on the West of the Iron Curtain, so the latter name became more known, see also Apt and Olderog [2019].

> Feeling uncomfortable about this lack of semantics, in 1974 I adopted the relational semantics then being advocated in CWI Amsterdam by Jaco de Bakker and Willem de Roever [1972], a single-sorted system of relations based on regular expressions, and used it to give Hoare's [. . .] language a semantics for a two-sorted system of programs and sentences of first order logic [. . .]

Formally, the syntax of dynamic logic consists of all first-order formulas (program variables permitted as terms), plus two modal operators: if S is a program and Q a dynamic logic formula, then $[S]\,Q$ and $\langle S \rangle\,Q$ are dynamic formulas, as well. As pointed out by Pratt, dynamic logic programs are modeled after relational calculus: In addition to assignment and sequential composition, there is nondeterministic choice "$S_0 \cup S_1$", finite, nondeterministic repetition "S^*", and test-else-fail "$S?$" (here written with a trailing question mark to distinguish tests syntactically from assertions). All the usual program statements, including guarded commands, are definable. Obviously, dynamic logic programs are nondeterministic.

Formula $[S]\,Q$ is true when any terminating run of S ends in a state satisfying Q. In other words, $[S]\,Q$ holds if and only if wlp.S.Q does; the Hoare triple (6.1) is true if and only if $P \implies [S]\,Q$ holds. Both relations were duly noted in Pratt's paper.[27] Dynamic logic is more expressive than Hoare logic, for example, program equivalence (relative to P and Q) can be expressed by means of the formula $(P \implies [S_0]\,Q) \iff (P \implies [S_1]\,Q)$.

The formula $\langle S \rangle\,Q$ is true when there is at least one terminating run of S that ends in a state satisfying Q. Both modalities are duals and related by $[S]\,Q \iff \neg\langle S \rangle \neg Q$. Hence, the dynamic logic modalities are *conjugates* in the sense of Dijkstra and Scholten [1990, chapter 6], whereas wp and wlp are conjugates only for deterministic programs by the equivalence. Indeed, since wp insists on termination for *all* runs, it is less natural to define and it has fewer nice properties than wlp.S; for example, it lacks distributivity over disjunction. Ultimately, this was also recognized by Dijkstra: while in DoP he still considers wp as being fundamental, in PCPS he had changed his mind and admits (page 130):

> Of the two, wp.S and wlp.S, the latter is the more fundamental one—there is no way of defining wlp.S in terms of wp.S—; it is also the one with the nicer properties [. . .]

Pratt viewed dynamic logic as a multi-modal logic with Kripke semantics, where each program S is associated with a modality relating its pre- and post-states. From this point of view, given S, the "always" modality $\Box_S\,Q$ expresses that Q holds in all

[27] He wrote erroneously that the correspondence is to wp.S.Q, but this is not relevant here.

terminating states of S reachable from the current state. Dually, the "sometimes" modality $\Diamond_S Q$ expresses that there is at least one terminating state of S reachable from the current state. $[S]\,Q$ and $\langle S\rangle\,Q$ are simply a notational variant of $\Box_S Q$ and $\Diamond_S Q$, respectively. Each state is evaluated with respect to a (possibly different) first-order model.

We mentioned above that both Hoare logic and the wp-calculus can be easily modeled in dynamic logic. In fact, Harel and Pratt [1978] completely formalized the wp-calculus in dynamic logic and formally re-proved Dijkstra's main results. They also provided a dynamic logic version of symbolic execution [Harel et al. 1977], formalizing the approach of Burstall [1974]. For example, a strongest postcondition rule for assignment can be proven as a derived rule in dynamic logic. In addition to an invariant rule, similar to (6.4), a large variety of induction rules with differing strength were suggested [Harel et al. 2000]. To summarize, dynamic logic is a general framework for a number of program verification approaches, see Table 6.1. But what did Dijkstra think of it?

To establish laws about programming languages, he saw the need to perform inferences over Hoare triples [Dijkstra 1989b], that is, the need for a logic closed over Hoare triples, but he does not mention dynamic logic in this context. There is a single, indirect mention of dynamic logic in all the EWD notes: in October 1977 it is referred to as a "modal logic" [Dijkstra 1977b] during a conversation with Harel about a recent paper Harel had written with his Ph.D. advisor Pratt. The discussion seemed not to have gone well. In Harel and Pratt [1978] there is a remarkable passage:

> We are strongly against the approach implicit in Dijkstra's work, in which the basic construct (wp) and the properties required of it, appear to obscure the simple parts of which it consists. We are further against the attempt to define the semantics of a language using a complex (execution-method dependent in this case) notion.

A harsher rebuke, accusing Dijkstra to be an offender against two of his most revered principles—simplicity and nonoperational description—is hardly

Table 6.1 **Comparison of different program verification frameworks**

	Hoare logic	wp-calculus	symbolic execution	dynamic logic
control flow	Any	backward	forward	any
deterministic	Yes	no	agnostic	no
syntax closed	No	yes	no	yes
loops	Invariants	invariants	invariants/induction	agnostic

imaginable. It seems highly unlikely that Dijkstra was unaware of this passage, prominently published in the POPL 1978 proceedings. Still, I could not find any direct repercussion in Dijkstra's writings, but for sure his general skepticism of what he perceived as "theory" (see beginning of Section 6.2) was reinforced.

Even today, dynamic logic is mostly ignored in the programming language community, who otherwise embraces formalization in the meantime. The contrasting reception of dynamic logic in the late 1970s by theoretical computer scientists (enthusiastic) and programming language people ("superfluous") is mentioned in Pratt [2017], who fails, however, to point out that he and Harel had stoked the fire. Possibly, the success of dynamic logic in theoretical computer science even had a detrimental effect on its view by practitioners of program verification. Certainly, the kind of Hilbert-style axiomatization used in Harel [1984] is not useful for program verification. An early attempt to implement a verification tool based on dynamic logic had little impact [Litvintchouk and Pratt 1977]. Only later was dynamic logic axiomatized in a Gentzen-style *sequent calculus* suitable for automated deduction [Hähnle et al. 1986] and subsequently used to perform program verification with symbolic execution [Heisel et al. 1987]. By then verification condition generation and Hoare logic were firmly entrenched as tools in the program verification community.

In summary, symbolic execution, Hoare logic, and weakest precondition reasoning are closely related and can be described with dynamic logic. It is the differences in the pragmatic usage of these formalisms that distinguishes them, less so their theoretical underpinnings.

6.3 From Program Correctness to Program Certification

Before I make an attempt to appraise Dijkstra's legacy in the field of program verification, I would like to trace some of the developments and trajectories program verification took in the last decades.

6.3.1 Understanding and "Understanding"

Dijkstra was interested in developing correct programs, or proving programs correct, but this is different than having a mechanically verified proof of correctness. So the question is: what are the challenges to deal with this step? On one hand, Dijkstra [1988, 1996] enthusiastically endorsed "manipulating uninterpreted formulae":

> [. . .] while from an operational point of view a program can be nothing but an abstract symbol manipulator, the designer had better regard the program

> as a sophisticated formula. And we also know that there is only one trustworthy way for the design of sophisticated formulae, viz. derivation by means of symbol manipulation. We have to let the symbols do the work, for that is the only known technique that scales up.

But this does not imply he endorsed *automation*. He feared automated theorem proving would cause "losing all interest in the simplification of doing formal mathematics" and maintained that "my calculational proofs [are] shorter than any alternative I can think of" [Dijkstra 1988]. In Dijkstra [1982e] a fundamental objection against automated proofs is raised:

> To the idea that proofs are so boring that we cannot rely upon them unless they are checked mechanically I have nearly philosophical objections, for I consider mathematical proofs as a reflection of my understanding and "understanding" is something we cannot delegate, neither to another person, nor to a machine.

Dijkstra was aware of the fact that his insistence on being able to "understand" each step in a verification proof forces one to limit the capabilities of the verification target. The following passage [Dijkstra 1982f] makes his stance clear:

> [De] Bakker [. . .] proudly demonstrated a proof rule for the PASCAL procedure call without the restrictions on the parameters that Igarashi, London and Luckham had introduced. As his new proof rule has the consequence that one cannot prove the correctness of the procedure in isolation, but may in principle need different proofs for the different calls, any reasonable man would conclude that the Igarashi–London–Luckham restriction is such a wise one that we may speak of a flaw in the design of PASCAL. But de Bakker emphatically refused to draw that conclusion!

In other words: if it is necessary to restrict the programming language to retain full "understanding" of verification proofs—then so be it.

In my opinion, Dijkstra could have afforded to be less restrictive without abandoning his central tenet that "'understanding' is something we cannot delegate": Yes, to verify "real" programs, we need to rely heavily on automation. But does this imply we let a machine do the "understanding?" It is hard for me to fathom what speaks against a human supplying the main ideas for a proof—the ones that require "understanding"—in the form of invariants and auxiliary assertions and let the machine do the tedious groundwork of simplification, checking the corner cases, and so on. It is simply unnecessary and *does not increase understanding* structure and mathematical content of a long proof to tediously check each micro step, each substitution, each application of *modus ponens*.

In contrast, this is how mechanical program verification is performed today [Hähnle and Huisman 2019]: it is cast as a *logical deduction* problem so as to benefit from the vast progress made in automatic formula simplification during the last years [Moura and Bjørner 2008, Barrett et al. 2011]. To appreciate this *logical turn* in program verification, it is instructive to look at the technical reasons why it is advantageous to view program verification as a logical deduction problem:

1. Axioms and rules must be *schematic*, so they can be instantiated with infinitely many assertions and programs. This makes it possible to represent the semantics of an infinite language with finitely many rules.
2. Assertions are built from *symbolic* expressions, so it becomes possible to reason about all possible initial values, hence, about all possible runs of a program.
3. State change—for example, rules (6.2), (6.6)—is recorded with syntactic *substitution*.
4. The semantics of complex statements—such as rules (6.3), (6.7), (6.8)—is modeled by proof *composition*: the proofs of the premises are composed into a proof of the conclusion.

These four aspects are characteristic of calculi for deduction in symbolic logic, and the machinery of automated deduction systems has been optimized during decades to implement exactly those aspects with maximum efficiency. But the fact that logic became a *lingua franca* in program specification and verification is not just down to technical progress in automated deduction. Equally important is a highly standardized (up to notational conventions) syntax and an unambiguous, model theoretic semantics. This is the prerequisite of "interoperability" in program verification between people [Huisman et al. 2020] as well as verification tools [Barrett et al. 2010].

6.3.2 Program Verification at Scale

Dijkstra concentrated on verification of stand-alone algorithms, but even a programming language designed for ease of specification and verification, such as Eiffel [Meyer 1997], needs a library of routines to be usable in practice. To verify even a small system, it is necessary to break down the task into manageable units.

Two chief possibilities to achieve modularity in program verification emerged (A) refinement and (B) composition.[28]

The main limitation of (A) is that the design of the target language has to be carefully controlled and its features frozen, which excludes programming languages designed "in the wild." Even though the wp-calculus does not directly support program development by refinement, it is very much in its (cbc) spirit. Refinement calculi were pioneered by Wirth [1971]. The approaches of Back [1981], Gries [1981], and Kourie and Watson [2012] all directly refer to Dijkstra's work. Less directly related are the B [Abrial 1996] and Event-B [Abrial 2010] refinement frameworks. The refinement calculus of Morgan [1990], also based on wp-reasoning, contributes the paradigm that a program can be viewed as the implementation of a *contract* with its prospective user. The contract-centric view was central in the specification proposal of Meyer [1992] and the design of Eiffel.

In fact, to realize option (B), procedure or method contracts became the dominant technique in program verification [Hähnle and Huisman 2019, Gurov et al. 2020]. The reason is that procedures (methods, functions, routines, . . .) arguably are *the* central concept to structure any program longer than one or two dozen lines. Because contract-based specification is exactly aligned with the procedure-based programming model, contracts are a natural choice to decompose a verification task. Contract-based program verification is very much in line with Dijkstra's suggestion to master complexity (of program verification) by subdivision or *Divide et impera* [Dijkstra 1979]:

> [The programmer] makes the complete specifications of the individual parts [. . .] he satisfies himself that the total problem is solved provided he had at his disposal program parts meeting the various specifications.

Importantly, the specifications need to satisfy the *Principle of Non-interference* of Dijkstra [1979] that "the correct working of the whole can be established by taking, of the parts, into account their exterior specification only" (discussed in greater depth in Chapter 7 by David Gries). But the quote above about de Bakker indicates Dijkstra in fact had an even stronger restriction in mind that would allow one to compose specifications in a completely local manner: To specify and verify a given program segment it should not be necessary to look at program points outside of it. Or, at least, any required information, such as global variables, is

[28] There might be another path in addition to (A) and (B) for breaking down a complex task: (C) program transformation or refactoring. One might start with a perfectly executable but naïve version of a program and gradually transform it into a more efficient one while keeping the weakest preconditions invariant. Refactoring is familiar to programmers [Fowler 1999] but moved only recently into the focus of program verification [Steinhöfel 2020].

easily statically obtainable. This makes it difficult to handle programming language features whose runtime semantics cannot be determined statically. To me, this seems too restrictive: Not all such features are whimsical and can be avoided without compromising usability. It is difficult to imagine a widely used imperative programming language completely without reference types, encapsulation, or error handling. But such features cannot be specified completely locally [Hatcliff et al. 2012]. A prominent case is the frame problem: which memory locations could have been affected as the side effect of executing a given piece of code? For Dijkstra, this might have been a case of "complexity of our own making," to be avoided by restraint in language design. But should we really just refuse to verify the many safety-critical JAVA, Python, or C programs out there? The attitude of the program verification community today is that this would mean to sell short the potential and impact of program verification.

6.3.3 The VCG Architecture

A *verification condition generator* (VCG) reduces a given program and postcondition (plus loop invariants) to a set of first-order formulas—verification conditions—whose validity is sufficient to imply correctness. Verification conditions can be discharged with a sufficiently powerful first-order theorem prover. The architecture of such a program verification system was first described by Igarashi et al. [1974] based on Hoare logic. Moriconi and Schwartz [1981] observed that a VCG can be seen as a predicate transformer and that this allows to argue its soundness. Ever since, the VCG architectural style of program verification systems has been closely associated with Dijkstra's wp-calculus. It proved to be hugely popular and successful, in particular, in combination with two improvements:

(1) A high-level target programming language is compiled to a minimalist intermediate language [Barnett et al. 2006, Marché 2007] before VCG is performed. The advantage is that the low-level (intermediate) language is stable and has only a few, simple instructions.[29] Therefore, wp-reasoning is a good choice for this kind of VCG architecture.

(2) The rise of SMT (for: *satisfiability modulo theories*) solvers [Moura and Bjørner 2008, Barrett et al. 2011] to automatically prove first-order verification conditions over theories describing program expressions, such as integer arithmetic, arrays, bitvectors, and so on. In fact, the development of SMT solvers was *driven* by the needs of VCG-style verification.

[29] Intermediate languages would not necessarily have been to Dijkstra's taste, though, because they feature jumps.

There is a direct lineage from Dijkstra to modern VCG architectures via Nelson [1989] and Leino [Detlefs et al. 1998]. Some of the world's most powerful contemporary deductive verification systems have a VCG architecture [Barnett et al. 2006, Filliâtre and Marché 2007, Cohen et al. 2009, Leino and Wüstholz 2014]. Compilation to intermediate languages can be rather complex and may involve several layers [Blom and Huisman 2014]. In consequence, one has to pay close attention to the correctness of the translation.

The affinity between Dijkstra's wp-calculus and VCG-style program verifiers is doubly ironic: First, VCG constitutes an *operational* reading of the wp-calculus, and second, these systems work in "auto-active" mode [Tschannen et al. 2015] or even in batch mode on intermediate languages—quite the opposite of what Dijkstra had in mind for program verification.

6.3.4 Going Backwards by Forward Reasoning

A slightly less popular architecture than VCG, but also highly effective and successful [Balser et al. 2000, Jacobs and Piessens 2008, Ahrendt et al. 2016], is based on symbolic execution (as discussed in Section 6.2.5). The approach realized in the verification tool KeY [Ahrendt et al. 2016] is of particular interest here because it combines forward symbolic execution with wp-reasoning. This is possible due to a re-formulation of the state change rules (6.2) and (6.6). Instead of computing the weakest precondition of an assignment right away, the assignment's effect is merely recorded and applied lazily. In dynamic logic syntax (Section 6.2.7):

$$\frac{P \Longrightarrow \{x := E\}\,(\langle S\rangle\, Q)}{P \Longrightarrow \langle x := E;\; S\rangle\, Q}\,.$$

Expression $\{x := E\}$ in the premise is called an *update* and can be read as an explicit substitution to be (*eventually*) applied to $\langle S\rangle\, Q$.[30] More generally, the meaning of a formula $\mathcal{U}\,P$ is that P is evaluated in the state that results from applying the change expressed by the update $\mathcal{U}$ to the current state. For example, the formula $\{x := x + y\}(x = y)$ is true in any state in which $y = 0$. Delayed application of the effect of state changes makes it possible to symbolically execute a program in the forward direction without having to compute the strongest postcondition. Rules are applied to formulas of the form $\mathcal{U}\,(\langle S_0;\; S_1\rangle\, Q)$, resulting in the premises of the form $\mathcal{U}\mathcal{U}'\,(\langle S_1\rangle\, Q)$. This process continues until the program inside the modality

[30] Observe the similarity to the notation of Dijkstra and Scholten [1990] for substitutions, see rule (6.2).

is fully symbolically executed and is of the form $\mathcal{U}_0 \cdots \mathcal{U}_k\,(\langle\,\rangle\, Q)$, at which point the updates are finally applied to the first-order postcondition Q, resulting in wp.S.Q.[31]

But what about the path explosion problem mentioned above? Two tactics mitigate this issue: first, updates are applied lazily but are simplified *eagerly*. For example, the update in formula $\{x := E\}\,(\langle S\rangle\, Q)$ can be discarded, whenever x is not read in S and does not occur in Q. Second, symbolic execution paths with only minor differences in states and path conditions can be merged [Scheurer et al. 2016]. It has been experimentally demonstrated that lazy update simplification and eager simplification, in connection with path merging techniques, control the path explosion problem to the extent that verification of highly complex algorithms with symbolic execution becomes feasible [De Gouw et al. 2019].

6.4 The Future of the Legacy

I structure this final section along some main themes that have surfaced up to now: weakest preconditions, theory versus practice, phv versus cbc, and simplicity.

6.4.1 Weakest Precondition Reasoning

To think in terms of weakest preconditions, computed by predicate transformers, when arguing the correctness of programs became one of the central strategies in program verification and it is Dijkstra's lasting merit to have stressed and propagated this thinking by virtue of his wp-calculus. Originally intended as a programming language semantics, wp-based predicate transformers nowadays are mostly seen as the core part of a VCG, where the precise fixpoint semantics of loops is approximated by invariants.

Whenever the task of designing a proof system for a new computational concept arises, it became a natural approach to start out from wp-reasoning. Among the many contributions along these lines, we only mention fair schedulers [Broy and Nelson 1994] and probabilistic programs [McIver and Morgan 2005, Kaminski et al. 2018] (where the weakest preconditions become weakest pre-expectations).

It is also interesting to note that Dijkstra's recursive construction of predicate transformers for loops by incremental execution of the loop body is reminiscent of SAT-based model checking, where loops are gradually unwound (and termination is guaranteed by the fact that the considered verified systems are finite) [Bjesse and Claessen 2000].

[31] Depending on how strong the chosen loop invariants are, the result might only imply the weakest precondition. For straight-line programs it is exact.

6.4.2 Theory and Practice of Program Verification

It is unfortunate that Dijkstra's negative views on what he perceived as Theoretical Computer Science led him to dismiss concepts relevant to his own endeavor, such as dynamic logic and relational semantics.

From today's point of view, a reconciliation between theory and practice seems to have taken place in program verification. To work successfully at the cutting edge of program verification research, it is necessary to have a solid grasp on theoretical issues. Papers published at leading program verification venues, such as CAV, IJCAR, or the *Journal of Automated Reasoning*, have considerable mathematical depth. "Theoretical" topics in model theory, type theory, proof theory, and complexity suddenly come to the forefront during attempts to solve practical issues such as computational feasibility.

On the other hand, verification approaches dealing with "toy" languages and problems hardly stand a chance to get published. Papers at major programming language conferences often are accompanied by a mechanization done with a proof assistant based on constructive type theory [Nipkow et al. 2002, Bertot and Castéran 2004]. Executable system code [Alkassar et al. 2010, Heiser et al. 2020] and complex library functions [De Gouw et al. 2014, 2019, Hiep et al. 2020] were fully verified. Even complete *application programming interfaces* of commercial programming languages were fully specified and verified. Among the first was a dialect for mobile devices of the Java language called Java Card,[32] performed with the deductive verification tool KeY [Mostowski 2007]. A further breakthrough was the verification of the container library of the Eiffel language [Polikarpova et al. 2018]. The company AdaCore[33] offers commercial program verification for an Ada subset. Major IT companies use program verification tools to improve software and service quality [Ball et al. 2004, Distefano et al. 2019], and some of the most brilliant researchers in program verification are now working at Microsoft, Amazon Web Services, or Facebook.

6.4.3 phv and cbc Revisited

The deductive verification community focused on *post hoc* verification [Hähnle and Huisman 2019]. Despite Dijkstra's misgivings, considerable progress was achieved, as pointed out above. Still, Dijkstra's worries were justified, as it turned out that to discover a provable specification for a given program in hindsight is not impossible but is a very hard problem, and the effort to come up with a suitable specification

[32] https://www.oracle.com/java/technologies/java-card-tech.html.

[33] https://www.adacore.com/.

often exceeds the effort of verification several times. In addition, the resulting specifications are bulky and hard to understand [Baumann et al. 2012]—the opposite of Dijkstra's call for beauty and simplicity!

The tradition of correctness-by-construction, in the form of refinement-based calculi, has been kept alive [Kourie and Watson 2012], but it is not mainstream and as yet does not benefit in the same way from the progress in automated reasoning as phv does. However, recently, a promising collaboration between the cbc and the deductive verification communities was established [Runge et al. 2019, Knüppel et al. 2020], this is a development to keep an eye on.

In my view, the choice of phv versus cbc is largely orthogonal to the choice of a formal framework. The considerations in Section 6.2.7 suggest any deductive framework can be recast for performing cbc-style verification. The question is how exactly one can benefit from automation and what is the best way to set up the workflow to achieve maximum usability.

6.4.4 Towards a New *Discipline*

As pointed out in Section 6.3.1, I believe that Dijkstra was overly restrictive with his postulate to understand every step in a correctness proof, but his fight against willful complexity is now timelier than ever. Let me illustrate the issue with an anecdote.

In 2015, I was involved in detecting (and fixing) a serious bug in the default sorting algorithm TimSort[34] for collections used in the Open JDK, Android, Python, Haskell, Go, Apache, and some other languages and frameworks. It turned out that certain, rather long, input arrays would cause the program to abort with an uncaught top-level exception.[35] The bug was discovered—within hours—during an attempt at phv of the Java implementation of TimSort. The designers had supplied an invariant for the main loop, but the algorithm was complex enough to keep anyone from realizing that for some inputs the invariant is broken. Many years of usage and testing failed to uncover the bug—a classic case for the need of mechanical verification. The proof of the fixed code is very complex and consists of over 2,000,000 steps, of which over 99 % were computed automatically. A proof by hand is completely infeasible. But the structure, interaction points, invariants, and method contracts that had to be supplied required deep insight into the mathematical arguments that make the proof work.

[34] svn.python.org/projects/python/trunk/Objects/listsort.txt.

[35] See www.envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/ and De Gouw et al. [2019].

Does it mean all is fine and dandy in the world of semiautomated, deductive verification? Well, hardly! The TimSort algorithm is ingenious; however, its implementation is ugly. It was relentlessly optimized for (suspected) performance. The most complex methods exhibit convoluted control flow with nested loops, local jumps, numerous conditionals, and—best of all—behavior depending on the overflow semantics of finite integer types. Clearly, for ease of verification, TimSort should be implemented differently, but how? And how to ensure that performance does not degrade? *A posteriori* verification was more successful than Dijkstra could have imagined, while cbc, "developing the program and its correctness proof" not merely "hand in hand," but largely *by hand*, does not scale to this complexity.

Thus, Dijkstra's demand, cited in Section 6.1.1, that "the program should be designed in such a way that its correctness could, indeed, be established" is still unaddressed. There must be a middle ground, yet to be discovered—we need someone to write *A Discipline of Designing Programs Amenable to Verification*!

Acknowledgment

The following people provided feedback on drafts of this chapter that helped to improve the presentation: David Gries, Mia Indrika, Peter Schmitt, Christoph Walther. An anonymous reviewer discovered technical inaccuracies and pointed out Dijkstra [1989b] to me.

References

J.-R. Abrial. August. 1996. *The B Book: Assigning Programs to Meanings*. Cambridge University Press. DOI: https://doi.org/10.1017/CBO9780511624162.

J.-R. Abrial. 2010. *Modeling in Event-B—System and Software Engineering*. Cambridge University Press. DOI: https://doi.org/10.1017/CBO9781139195881.

W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich (Eds.). 2016. *Deductive Software Verification—The KeY Book: From Theory to Practice*, Vol. 10001: Lecture Notes in Computer Science. Springer. DOI: https://doi.org/10.1007/978-3-319-49812-6.

E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. 2010. Automated verification of a small hypervisor. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani (Eds.), *Verified Software: Theories, Tools, Experiments, Third Intl. Conference, VSTTE, Edinburgh, UK*, Vol. 6217: Lecture Notes in Computer Science. Springer, Berlin, 40–54. DOI: https://doi.org/10.1007/978-3-642-15057-9_3.

K. R. Apt. 1981. Ten years of Hoare's logic: A survey—Part 1. *ACM Trans. Program. Lang. Syst.* 3, 4, 431–483. DOI: https://doi.org/10.1145/357146.357150.

K. R. Apt and E. Olderog. 2019. Fifty years of Hoare's logic. *Form. Asp. Comput.* 31, 6, 751–807. DOI: https://doi.org/10.1007/s00165-019-00501-3.

R. Back. 1981. On correct refinement of programs. *J. Comput. Syst. Sci.* 23, 1, 49–68. DOI: https://doi.org/10.1016/0022-0000(81)90005-2.

J. W. D. Bakker, A. D. Bruin, and J. Zucker. 1980. *Mathematical Theory of Program Correctness*. Prentice-Hall International Series in Computer Science. Prentice Hall, Upper Saddle River, NJ.

R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv*. 51, 3, 50:1–50:39. DOI: https://doi.org/10.1145/3182657.

T. Ball, B. Cook, V. Levin, and S. K. Rajamani. 2004. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In E. A. Boiten, J. Derrick, and G. Smith (Eds.), *Proc. Integrated Formal Methods, 4th Intl. Conf., IFM 2004*, Canterbury, UK, Vol. 2999: Lecture Notes in Computer Science. Springer, 1–20. DOI: https://doi.org/10.1007/978-3-540-24756-2_1.

M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. 2000. Formal system development with KIV. In T. Maibaum (Ed.), *Fundamental Approaches to Software Engineering*, Vol. 1783: Lecture Notes in Computer Science. Springer-Verlag. DOI: https://doi.org/10.1007/3-540-46428-X_25.

M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. 2006. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever (Eds.), *Formal Methods for Components and Objects, 4th Intl. Symp. FMCO*, Amsterdam, The Netherlands, Revised Lectures, Vol. 4111: Lecture Notes in Computer Science. Springer, Berlin, 364–387. DOI: https://doi.org/10.1007/11804192_17.

C. W. Barrett, A. Stump, and C. Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening (Eds.), *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*. University of Iowa, Edinburgh, UK.

C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. 2011. CVC4. In G. Gopalakrishnan and S. Qadeer (Eds.), *Computer Aided Verification: 23rd Intl. Conf., CAV*, Snowbird, UT, USA, Vol. 6806: Lecture Notes in Computer Science. Springer, Berlin, 171–177. DOI: https://doi.org/10.1007/978-3-642-22110-1_14.

C. Baumann, B. Beckert, H. Blasum, and T. Bormer. 2012. Lessons learned from microkernel verification—Specification is the new bottleneck. In F. Cassez, R. Huuck, G. Klein, and B. Schlich (Eds.), *Proc. 7th Conference on Systems Software Verification*, Vol. 102: EPTCS. 18–Open Publishing Association, 18–32. DOI: https://doi.org/10.4204/EPTCS.102.4.

B. Beckert, S. Schlager, and P. H. Schmitt. 2005. An improved rule for while loops in deductive program verification. In K.-K. Lau (Ed.), *Proc. 7th International Conference on Formal Engineering Methods (ICFEM)*, Manchester, UK, Vol. 3785: Lecture Notes in Computer Science. Springer-Verlag, Berlin, 315–329. https://doi.org/10.1007/11576280_22.

Y. Bertot and P. Castéran. 2004. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer. DOI: https://doi.org/10.1007/978-3-662-07964-5.

P. Bjesse and K. Claessen. 2000. SAT-based verification without state space traversal. In W. A. H. Jr. and S. D. Johnson (Eds.), *Formal Methods in Computer-Aided Design, Third Intl.*

Conf. FMCAD, Austin, TX, Vol. 1954: Lecture Notes in Computer Science. Springer, Heidelberg, 409–426. DOI: https://doi.org/10.1007/3-540-40922-X_23.

A. Blass and Y. Gurevich. 2001. Inadequacy of computable loop invariants. *ACM Trans. Comput. Log.* 2, 1, 1–11. DOI: https://doi.org/10.1145/371282.371285.

S. Blom and M. Huisman. 2014. The VerCors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun (Eds.), *FM 2014: Formal Methods, 19th Intl. Symposium*, Singapore, Vol. 8442: Lecture Notes in Computer Science. Springer, 127–131. DOI: https://doi.org/10.1007/978-3-319-06410-9_9.

J. Bosch. 2016. Speed, data, and ecosystems: The future of software engineering. *IEEE Softw*. 33, 1, 82–88. DOI: https://doi.org/10.1109/MS.2016.14.

R. S. Boyer, B. Elspas, and K. N. Levitt. June. 1975. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Not*. 10, 6, 234–245. DOI: https://doi.org/10.1145/800027.808445.

M. Broy and G. Nelson. 1994. Adding fair choice to Dijkstra's calculus. *ACM Trans. Program. Lang. Syst*. 16, 3, 924–938. DOI: https://doi.org/10.1145/177492.177727.

R. Bubel and R. Hähnle. 2016. KeY-Hoare. In W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich (Eds.), *Deductive Software Verification—The KeY Book: From Theory to Practice*, chapter 17, Vol. 10001: Lecture Notes in Computer Science. Springer, 571–589. DOI: https://doi.org/10.1007/978-3-319-49812-6_17.

R. M. Burstall. 1974. Program proving as hand simulation with a little induction. In *Information Processing'74*. Elsevier/North-Holland, 308–312.

E. M. Clarke. 1979. Program invariants as fixedpoints. *IEEE Comput*. 21, 4, 273–294. DOI: https://doi.org/10.1007/BF02248730.

E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. 2009. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel (Eds.), *Theorem Proving in Higher Order Logics, 22nd Intl. Conf. TPHOLs*, Vol. 5674: Lecture Notes in Computer Science. Springer, 23–42. DOI: https://doi.org/10.1007/978-3-642-03359-9_2.

J. W. De Bakker and W. P. De Roever. 1972. A calculus for recursive program schemes. In M. Nivat (Ed.), *Proc. IRIA Symposium on Automata, Languages and Programming, Colloquium*, Paris, France. North-Holland, Amsterdam, 167–196.

S. De Gouw, F. S. De Boer, and J. Rot. 2014. Proof pearl: The KeY to correct and stable sorting. *J. Autom. Reason*. 53, 2, 129–139. DOI: https://doi.org/10.1007/s10817-013-9300-y.

S. De Gouw, F. S. De Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel. 2019. Verifying OpenJDK's sort method for generic collections. *J. Autom. Reason*. 62, 6, 93–126. DOI: https://doi.org/10.1007/s10817-017-9426-4.

R. A. DeMillo, R. J. Lipton, and A. J. Perlis. 1977. Social processes and proofs of theorems and programs. In R. M. Graham, M. A. Harrison, and R. Sethi (Eds.), *Conf. Record of the Fourth ACM Symp. on Principles of Programming Languages*, Los Angeles, CA. ACM, 206–214. DOI: https://doi.org/10.1145/512950.512970.

D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. December. 1998. *Extended Static Checking*. Research Report #1998-159. Compaq Systems Research Center, Palo Alto. https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-159.pdf.

L. Deutsch. May. 1973. *An Interactive Program Verifier*. Ph.D. thesis. Dept. of Computer Sci., U. of California, Berkeley. http://www.softwarepreservation.org/projects/verification/pivot/Deutsch-An_Interactive_Theorem_Prover-1973.pdf.

E. W. Dijkstra. 1968. Go To statement considered harmful. *Commun. ACM* 11, 3, 147–148. Letter to the Editor. DOI: https://doi.org/10.1145/362929.362947.

E. W. Dijkstra. 1974a. Trip report E. W. Dijkstra, Edinburgh and Newcastle, 1–6 September 1974. EWD448. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD448.PDF.

E. W. Dijkstra. 1974b. Some questions. EWD463. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD463.PDF.

E. W. Dijkstra. 1976a. *A Discipline of Programming*. Prentice-Hall.

E. W. Dijkstra. 1976b. Programming: From craft to scientific discipline. EWD566. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd05xx/EWD566.PDF.

E. W. Dijkstra. 1977a. A political pamphlet from the Middle Ages. EWD638. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD638.PDF.

E. W. Dijkstra. 1977b. Trip report E. W. Dijkstra, USA 19–30 October, 1977. EWD645. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD645.PDF.

E. W. Dijkstra. 1979. Programming considered as a human activity. In E. N. Yourdon (Ed.), *Classics in Software Engineering*. Yourdon Press, Upper Saddle River, NJ, 1–9. Reprint of EWD117.

E. W. Dijkstra. 1982a. *Selected Writings on Computing: A Personal Perspective*. Springer, New York, NY. DOI: https://doi.org/10.1007/978-1-4612-5695-3.

E. W. Dijkstra. 1982b. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer, New York, NY, 60–66. DOI: https://doi.org/10.1007/978-1-4612-5695-3_12.

E. W. Dijkstra. 1982c. Craftsman or Scientist? In *Selected Writings on Computing: A Personal Perspective*. Springer, New York, NY, 104–109. DOI: https://doi.org/10.1007/978-1-4612-5695-3_19.

E. W. Dijkstra. 1982d. Trip report E. W. Dijkstra 16th April/7th May, 1975, U.S.A. and Canada. In *Selected Writings on Computing: A Personal Perspective*. Springer, New York, NY, 120–128. DOI: https://doi.org/10.1007/978-1-4612-5695-3_21.

E. W. Dijkstra. 1982e. Formal techniques and sizeable programs. In *Selected Writings on Computing: A Personal Perspective*. Springer, New York, NY, 205–214. DOI: https://doi.org/10.1007/978-1-4612-5695-3_35.

E. W. Dijkstra. 1982f. Trip report E. W. Dijkstra, Poland and USSR, 4–25 September, 1976. In *Selected Writings on Computing: A Personal Perspective*. Springer, New York, NY, 235–244. DOI: https://doi.org/10.1007/978-1-4612-5695-3_43.

E. W. Dijkstra. 1988. A new science, from birth to maturity. In *ETH Zürich: 20-year Anniversary of Institut für Informatik*. ETH Zürich, Institut für Informatik. https://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1024.PDF. Also available as EWD1024.

E. W. Dijkstra. 1989a. The next forty years. EWD1051. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1051.PDF.

E. W. Dijkstra. 1989b. "Predicate Calculus and Program Semantics," fall 1989. EWD1060. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1060.PDF.

E. W. Dijkstra. 1995. Why American Computing Science seems incurable. Circulated privately. EWD1209. http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1209.PDF.

E. W. Dijkstra. 1996. The next fifty years. EWD1243a. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1243a.PDF.

E. W. Dijkstra. 1998. Society's role in mathematics. EWD1277. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1277.PDF.

E. W. Dijkstra. n.d. The moral of EWD237–EWD239. EWD240. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD240.PDF.

E. W. Dijkstra and C. S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer, New York. DOI: https://doi.org/10.1007/978-1-4612-3228-5.

D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8, 62–70. DOI: https://doi.org/10.1145/3338112.

M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Syst. J.* 15, 3, 182–211. DOI: https://doi.org/10.1147/sj.153.0182.

J.-C. Filliâtre and C. Marché. 2007. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns (Eds.), *Computer Aided Verification, 19th International Conference*, Berlin, Germany, Vol. 4590: Lecture Notes in Computer Science. Springer, 173–177.

R. W. Floyd. 1967. Assigning meanings to programs. In J. T. Schwartz (Ed.), *Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics,* Vol. 19. American Mathematical Society, Providence, RI, 19–32.

C. Fournet, C. Keller, and V. Laporte. 2016. A certified compiler for verifiable computing. In *IEEE 29th Computer Security Foundations Symposium, CSF*, Lisbon, Portugal. IEEE Computer Society, Los Alamitos, CA, 268–280. DOI: https://doi.org/10.1109/CSF.2016.26.

M. Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts.

C. A. Furia, B. Meyer, and S. Velder. 2014. Loop invariants: Analysis, classification, and examples. *ACM Comput. Surv.* 46, 3, Article 34, 1–51. DOI: https://doi.org/10.1145/2506375.

S. Galand and G. Loncour. 1975. Structured implementation of symbolic execution: A first part in a program verifier. *Inf. Process. Lett.* 3, 4, 97–103. DOI: https://doi.org/10.1016/0020-0190(75)90041-1.

D. Gries. 1981. *The Science of Programming*. Texts and Monographs in Computer Science. Springer. DOI: https://doi.org/10.1007/978-1-4612-5983-1.

D. Gurov, R. Hähnle, and E. Kamburjan. October. 2020. Who carries the burden of modularity? Introduction to ISoLA 2020 track on modularity and (de-)composition in verification. In T. Margaria and B. Steffen (Eds.), *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA*, Vol. 12476: Lecture Notes in Computer Science. Springer, Cham, 3–21. DOI: https://doi.org/10.1007/978-3-030-61362-4_1.

R. Hähnle and M. Huisman. 2019. Deductive verification: From pen-and-paper proofs to industrial tools. In B. Steffen and G. Woeginger (Eds.), *Computing and Software Science: State of the Art and Perspectives*, Vol. 10000: Lecture Notes in Computer Science. Springer, Cham, 345–373. DOI: https://doi.org/10.1007/978-3-319-91908-9_18.

R. Hähnle, M. Heisel, W. Reif, and W. Stephan. 1986. An interactive verification system based on dynamic logic. In J. Siekmann (Ed.), *Proc. 8th Conference on Automated Deduction CADE*, Oxford, Vol. 230: Lecture Notes in Computer Science. Springer, Heidelberg, 306–315. DOI: https://doi.org/10.1007/3-540-16780-3_99.

D. Harel. 1984. Dynamic logic. In D. Gabbay and F. Guenthner (Eds.), *Handbook of Philosophical Logic.* Vol. II: Extensions of Classical Logic, chapter 10. Springer, Dordrecht, 497–604. DOI: https://doi.org/10.1007/978-94-009-6259-0_10.

D. Harel and V. R. Pratt. 1978. Nondeterminism in logics of programs. In A. V. Aho, S. N. Zilles, and T. G. Szymanski (Eds.), *Conf. Record of the Fifth Annual ACM Symp. on Principles of Programming Languages*, Tucson, AZ. ACM Press, New York, NY, 203–213. DOI: https://doi.org/10.1145/512760.512782.

D. Harel, A. R. Meyer, and V. R. Pratt. 1977. Computability and completeness in logics of programs (preliminary report). In J. E. Hopcroft, E. P. Friedman, and M. A. Harrison (Eds.), *Proc. 9th Annual ACM Symp. on Theory of Computing*, Boulder, CO. ACM, New York, NY, 261–268. DOI: https://doi.org/10.1145/800105.803416.

D. Harel, D. Kozen, and J. Tiuryn. 2000. *Dynamic Logic*. Foundations of Computing. MIT Press. Cambridge, MA. DOI: https://dl.acm.org/doi/10.5555/557365.

J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. J. Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv*. 44, 3, Article 16, 1–58. DOI: https://doi.org/10.1145/2187671.2187678.

M. Heisel, W. Reif, and W. Stephan. 1987. Program verification by symbolic execution and induction. In K. Morik (Ed.), *Proc. 11th German Workshop on Artificial Intelligence*, *Informatik Fachberichte*, Vol. 152. Springer, Heidelberg, 201–210. DOI: https://doi.org/10.1007/978-3-642-73005-4_22.

G. Heiser, G. Klein, and J. Andronick. 2020. seL4 in Australia: From research to real-world trustworthy systems. *Commun. ACM* 63, 4, 72–75. DOI: https://doi.org/10.1145/3378426.

H. A. Hiep, O. Maathuis, J. Bian, F. S. D. Boer, M. C. J. D. van Eekelen, and S. D. Gouw. 2020. Verifying OpenJDK's LinkedList using KeY. In A. Biere and D. Parker (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems—26th Intl. Conf. TACAS*, Dublin, Ireland, Part II, Vol. 12079: Lecture Notes in Computer Science. Springer, Cham, 217–234. DOI: https://doi.org/10.1007/978-3-030-45237-7_13.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10, 576–580, 583. DOI: https://doi.org/10.1145/363235.363259.

C. A. R. Hoare and N. Wirth. 1973. An axiomatic definition of the programming language PASCAL. *Acta Inform.* 2, 4, 335–355. DOI: https://doi.org/10.1007/BF00289504.

M. Huisman, R. E. Monti, M. Ulbrich, and A. Weigl. 2020. The VerifyThis Collaborative Long Term Challenge. In W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, and M. Ulbrich (Eds.), *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, Vol. 12345: Lecture Notes in Computer Science. Springer, Cham, 246–260. DOI: https://doi.org/10.1007/978-3-030-64354-6_10.

S. Igarashi, R. L. London, and D. C. Luckham. 1974. Automatic program verification: I. A logical basis and its implementation. *Acta Inform.* 4, 145–182. DOI: https://doi.org/10.1007/BF00288746.

B. Jacobs and F. Piessens. August. 2008. The VeriFast program verifier. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven. http://www.cs.kuleuven.be/˜bartj/verifast/verifast.pdf.

B. L. Kaminski, J. Katoen, C. Matheja, and F. Olmedo. 2018. Weakest precondition reasoning for expected runtimes of randomized algorithms. *J. ACM* 65, 5, Article 30, 1–68. DOI: https://doi.org/10.1145/3208102.

J. C. King. 1969. *A Program Verifier*. Ph.D. thesis. Carnegie-Mellon University.

J. C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7, 385–394. DOI: https://doi.org/10.1145/360248.360252.

A. Knüppel, T. Runge, and I. Schaefer. 2020. Scaling correctness-by-construction. In T. Margaria and B. Steffen (Eds.), *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: 9th Intl. Symp. on Leveraging Applications of Formal Methods, ISoLA*, Rhodes, Greece, Part I, Vol. 12476: Lecture Notes in Computer Science. Springer, Cham, 187–207. DOI: https://doi.org/10.1007/978-3-030-61362-4_10.

K. Kojima and A. Igarashi. 2013. A Hoare logic for SIMT programs. In C. Shan (Ed.), *Programming Languages and Systems: 11th Asian Symp. APLAS*, Melbourne, Australia, Vol. 8301: Lecture Notes in Computer Science. Springer, Cham, 58–73. DOI: https://doi.org/10.1007/978-3-319-03542-0_5.

D. G. Kourie and B. W. Watson. 2012. *The Correctness-by-Construction Approach to Programming*. Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/978-3-642-27919-5.

S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan. 2010. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *Proc. 19th IEEE Asian Test Symposium, ATS*, Shanghai, China. IEEE Computer Society, Los Alamitos, 59–64. DOI: https://doi.org/10.1109/ATS.2010.19.

K. R. M. Leino and V. Wüstholz. 2014. The Dafny integrated development environment. In C. Dubois, D. Giannakopoulou, and D. Méry (Eds.), *Proc. 1st Workshop on Formal Integrated Development Environment, F-IDE*, Grenoble, France, EPTCS 149. Open Publishing Association, 3–15. DOI: https://doi.org/10.48550/arXiv.1404.6602.

S. D. Litvintchouk and V. R. Pratt. 1977. A proof-checker for dynamic logic. In R. Reddy (Ed.), *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, Cambridge, MA. William Kaufmann, San Mateo, CA, 552–558.

C. Marché. 2007. Jessie: An intermediate language for Java and C verification. In A. Stump and H. Xi (Eds.), *ACM Workshop on Programming Languages Meets Program Verification,* *PLPV*, Freiburg, Germany. ACM, New York, NY, 1–2. DOI: https://doi.org/10.1145/1292597.1292598.

J. McCarthy. 1966. A formal description of a subset of ALGOL. In T. B. Steel, Jr. (Ed.), *Formal Language Description Languages for Computer Programming.* North-Holland, Amsterdam, 1–12.

A. McIver and C. Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, New York, NY. DOI: https://doi.org/10.1007/b138392.

B. Meyer. 1992. Applying "design by contract." *Computer* 25, 10, 40–51. DOI: https://doi.org/10.1109/2.161279.

B. Meyer. 1997. *Object-Oriented Software Construction* (2nd. ed.). Prentice-Hall, Englewood Cliffs, NJ.

C. Morgan. 1990. *Programming from Specifications*. Intl. Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

M. Moriconi and R. L. Schwartz. 1981. Automatic construction of verification condition generators from Hoare logics. In S. Even and O. Kariv (Eds.), *Automata, Languages and Programming, 8th Colloquium*, Acre (Akko), Israel, Vol. 115: Lecture Notes in Computer Science. Springer, Heidelberg, 363–377. DOI: https://doi.org/10.1007/3-540-10843-2_30.

W. Mostowski. 2007. Fully verified Java Card API reference implementation. In B. Beckert (Ed.), *Proc. 4th Intl. Verification Workshop in Connection with CADE-21*, Bremen, Germany, CEUR Workshop Proceedings, Vol. 259. CEUR-WS.org. DOI: https://hdl.handle.net/2066/34755.

L. M. D. Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 14th Intl. Conf., TACAS*, Budapest, Hungary, Vol. 4963: Lecture Notes in Computer Science. Springer, Berlin, 337–340. DOI: https://doi.org/10.1007/978-3-540-78800-3_24.

G. Nelson. 1989. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.* 11, 4, 517–561. DOI: https://doi.org/10.1145/69558.69559.

T. Nipkow, L. C. Paulson, and M. Wenzel. 2002. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic,* Vol. 2283: Lecture Notes in Computer Science. Springer-Verlag, Berlin. DOI: https://doi.org/10.1007/3-540-45949-9.

G. D. Plotkin. 1979. Dijkstra's predicate transformers & Smyth's powerdomains. In D. Bjørner (Ed.), *Abstract Software Specifications, Copenhagen Winter School*, Vol. 86: Lecture Notes in Computer Science. Springer, Heidelberg, 527–553. DOI: https://doi.org/10.1007/3-540-10007-5_48.

N. Polikarpova, J. Tschannen, and C. A. Furia. 2018. A fully verified container library. *Form. Asp. Comput.* 30, 5, 495–523. DOI: https://doi.org/10.1007/s00165-017-0435-1.

V. R. Pratt. 1976. Semantical considerations on Floyd–Hoare logic. In *17th Annual Symp. on Foundations of Computer Science*, Houston, TX. IEEE Computer Society, Los Alamitos, CA, 109–121. DOI: https://doi.org/10.1109/SFCS.1976.27.

V. R. Pratt. 2017. Dynamic logic: A personal perspective. In A. Madeira and M. R. F. Benevides (Eds.), *Dynamic Logic. New Trends and Applications: First Intl. Workshop, DALI*, Brasilia, Brazil, Vol. 10669: Lecture Notes in Computer Science. Springer, Cham, 153–170. DOI: https://doi.org/10.1007/978-3-319-73579-5_10.

C. Rauch, S. Goncharov, and L. Schröder. 2017. Generic Hoare logic for order-enriched effects with exceptions. In P. James and M. Roggenbach (Eds.), *Recent Trends in Algebraic Development Techniques—23rd IFIP WG 1.3 Intl. Workshop, WADT 2016*, Gregynog, UK, Vol. 10644: Lecture Notes in Computer Science. Springer, Cham, 208–222. DOI: https://doi.org/10.1007/978-3-319-72044-9_14.

T. Runge, I. Schaefer, L. Cleophas, T. Thüm, D. G. Kourie, and B. W. Watson. 2019. Tool support for correctness-by-construction. In R. Hähnle and W. M. P. van der Aalst (Eds.), *Fundamental Approaches to Software Engineering, 22nd Intl. Conf. FASE*, Prague, Czech Republic, Vol. 11424: Lecture Notes in Computer Science. Springer, Cham, 25–42. DOI: https://doi.org/10.1007/978-3-030-16722-6_2.

A. Salwicki. 1970. Formalized algorithmic languages. *Bull. Acad. Polon. Sci., Sér. Math. Astron. Phys.* 21, XVIII(5), 227–232.

D. Scheurer, R. Hähnle, and R. Bubel. 2016. A general lattice model for merging symbolic execution branches. In K. Ogata, M. Lawford, and S. Liu (Eds.), *Proc. 18th Intl. Conf. on Formal Engineering Methods (ICFEM)*, Tokyo, Japan, Vol. 10009: Lecture Notes in Computer Science. Springer, 57–73. DOI: https://doi.org/10.1007/978-3-319-47846-3_5.

D. Steinhöfel. 2020. REFINITY to model and prove program transformation rules. In B. C. d. S. Oliveira (Ed.), *Programming Languages and Systems, 18th Asian Symp., APLAS*, Fukuoka, Japan, Vol. 12470: Lecture Notes in Computer Science. Springer, Cham, 311–319. DOI: https://doi.org/10.1007/978-3-030-64437-6_16.

J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. 2015. AutoProof: Auto-active functional verification of object-oriented programs. In C. Baier and C. Tinelli (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems—21st International Conference, TACAS*, London, UK, Vol. 9035: Lecture Notes in Computer Science. Springer, 566–580. DOI: https://doi.org/10.1007/978-3-662-46681-0_53.

B. Wegbreit. 1973. Heuristic methods for mechanically deriving inductive assertions. In N. J. Nilsson (Ed.), *Proc. 3rd Intl. Joint Conf. on Artificial Intelligence*. Stanford, CA. Morgan Kaufmann, San Francisco, CA, 524–536. http://ijcai.org/Proceedings/73/Papers/055B.pdf.

N. Wirth. 1971. Program development by stepwise refinement. *Commun. ACM* 14, 4, 221–227. DOI: https://doi.org/10.1145/362575.362577.

H. Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1–3, 308–334. DOI: https://doi.org/10.1016/j.tcs.2006.12.036.

7 Development of Correct Programs

David Gries

It's August 1967, and the topic of formal program correctness is in the air. Peter Naur's paper on "Proofs of algorithms by general snapshots" [Naur 1966] appeared a year earlier, and Edsger W. Dijkstra knew about it, for he and Peter were close friends and colleagues. Bob Floyd's [1967] paper on "Assigning meanings to programs" and the paper by James Painter and John McCarthy [1967] on the "Correctness of a compiler for arithmetic expressions" were presented in the same symposium in 1967. All three papers were about formally proving programs correct.

Edsger was aware of these papers. However, in August 1967, he wrote EWD209, later published as Dijkstra [1968a], in which he took an entirely different approach. He tackled the problem of *developing* programs and their proofs hand-in-hand, with the proof ideas leading the way. This continual emphasis on *method* set Edsger apart from everyone else. It stayed with him all his life, first with developing correct programs and later with developing mathematical proofs —on mathematical methodology, which occupied a great deal of his later professional life. Here is Edsger's summary in EWD209:

> As an alternative to methods by which the correctness of given programs can be established a posteriori, this paper proposes to control the process of program generation such as to produce a priori correct programs. An example is treated to show the form that such a control then might take. This example comes from the field of parallel programming; the way in which it is treated is representative for the way in which a whole multiprogramming system has actually been constructed.

Note that last sentence. Edsger is saying that he (along with the team he led) used this approach in developing a whole multiprogramming system! (We'll discuss this more in Section 7.1.) Of course, Edsger wasn't talking about a formal proof as envisaged in the three papers mentioned above. Still, he *was* talking about developing some form of proof and program hand-in-hand.

In 1968 in EWD227 [Dijkstra 1982b], Edsger coined the term *Stepwise program construction*, well ahead of Niklaus Wirth's [1971] famous paper "Program development by stepwise refinement." As early as 1972, he began transforming Tony Hoare's [1969] axiomatic approach to program correctness into his predicate-transformer approach. He taught it in an advanced course in Grenoble in December 1972 EWD356 [Dijkstra 1972b]; he discussed it in EWD360 [Dijkstra 1973a]; and he ended up showing us in his research monograph *A Discipline of Programming* [Dijkstra 1976] not only how to develop loop invariants and such but how to *calculate* parts of a program. Finally, in the early 1980s, in order to be able to perform syntactic manipulations when developing programs, he began investigating the properties of equivalence ≡ in EWD842 [Dijkstra 1982e]. The result was a calculus based on equivalence and Leibniz's substitution of equals for equals rather than *modus ponens*. It was far more useful for syntactic manipulations than the proof systems developed by logicians. This work culminated in the book Dijkstra and Scholten [1990].

Many of us joined the *methodology* bandwagon, and we have been discussing *program development* and *proof development* rather than just *programs* and *proofs* ever since. It forms the basis of how some of us teach topics such as searching, sorting, and data structures. We teach programming, not programs. Edsger made this possible. He did the groundwork upon which everyone else's work in the field is based.

In this chapter, I will go into more detail about how and why Edsger achieved all this.

7.1 Recurring Themes

As I look back at Edsger's EWDs and my correspondence with him in preparing to write this chapter, four themes emerge that were crucial in all he did. These themes appeared over and over again in his work and, for me, sum up what drove him to succeed as he did.

(1) *Program correctness*. Program correctness was paramount for Edsger; it shaped everything he did. Initially, of course, correctness proofs were not based on a formal proof system, such as Tony Hoare's axiomatic basis for computer programming [Hoare 1969]. But read carefully what Edsger wrote in June 1968 in EWD196 about the operating system he built, published as Dijkstra [1968c]:

> We have found it is possible to design a refined multiprogramming system in such a way that its logical soundness can be proved a priori [. . .]. The only errors that showed up during testing were trivial coding errors (with [. . .] only one error per 500 instructions), each of them located with[in] 10 minutes [. . .] and [. . .] correspondingly easy to remedy. [. . .] the testing is not yet completed, but the resulting system will be guaranteed to be flawless.

Edsger was head of the six-person team that developed this operating system for the Dutch Electrologica X8 computer. The project was started at least 3–4 years earlier, so well before anyone thought about formal proof systems. Yet, Edsger said its logical soundness could be proved *a priori*, and this is not for a sequential program but an operating system, a concurrent program. I find this astounding. It is such a bold claim.

(2) *Method is just as important as the result*. Edsger was interested not just in proofs of program correctness but on learning how to develop proof and program hand-in-hand. Here's a quote from EWD209, published as Dijkstra [1968a]:

> As an alternative to methods by which the correctness of given programs can be established a posteriori, this paper proposes to control the process of program generation such as to produce a priori correct programs.

This emphasis on method led to his development of not only a calculus of weakest preconditions but also, at a later stage, to calculational logic. Finally, much of Edsger's later life was devoted to research in methodology in mathematics as well as computing.

(3) *The gap between program text and the computations it evokes*. Edsger mentions this gap often, and he looked for ways to make it easier to bridge the gap in both sequential and multiprogrammed environments. His arguments in his famous paper *Goto statements considered harmful* [Dijkstra 1968b] were based on bridging this gap. We'll return to this topic later.

(4) *The need for beauty, elegance, simplicity, and clarity*. Edsger spoke of these qualities often in terms of both programming languages and programs. The earliest I could find was in EWD32, written in 1962 [Dijkstra 1962a]. Edsger spoke about the programming language, compiler, and computer as a tool, as a whole. This tool should be charming and elegant, he said, even worthy of our love. He stressed that this was not a joke and that he was terribly serious

about it. He goes on to say that the greatest virtues a program could show were Elegance and Beauty.

7.2 What's a Proof?

We said in the previous section that Edsger's six-person team designed a multiprogramming system in such a way that its logical correctness was proved *a priori* and that the only errors that showed up during testing were trivial coding errors, all easy to locate and remedy. This was an operating system, written in the assembly language of the computer. At the time, there was no concept of what it meant for a program to be correct, let alone an axiomatic method allowing one to prove correctness by a computer program.

What, then, did Edsger mean by *proof* at the time? Quite likely, it was the same as that of a mathematical proof: a convincing argument, written in a mixture of mathematical (and programming) notation and Dutch (or other natural language). But what are the components of a proof of a program? For that, we need to look at later EWDs, in which he consciously began thinking about program proofs and how to construct them. We'll try to distill the essence of his writings on this topic, though for full impact, you'll have to read the EWDs yourself.

7.2.1 Divide and Conquer

In early 1965, Edsger wrote EWD117 [Dijkstra 1965]. It was for a talk, but EWD117 doesn't say when and where he gave it. He discusses the relation between mathematical proofs and proofs of program correctness, saying that,

> One can never guarantee that a proof [of a theorem] is correct, the best one can say, is: "I have not discovered any mistakes." We sometimes flatter ourselves with the idea of giving watertight proofs, but in fact we do nothing but make the correctness of our conclusions plausible.

And he was right about this. There have even been published proofs of theorems that, ten years later, were found to have a mistake. (It must be mentioned that, today, thousands of theorems have been proved by computer using proof systems such as Coq and NuPRL.)

Edsger goes on to say, in his typical style, that,

> The programmer's situation [in proving correctness of a program] is closely analogous to that of the pure mathematician, who develops a theory and proves results. [. . .] In spite of all its deficiencies, mathematical reasoning presents an outstanding model of how to grasp extremely complicated structures with a brain of limited capacity. [. . .] The technique of mastering complexity is known since ancient times: 'Divide and rule'. The analogy between

> proof construction and program construction is, again, striking [. . .] complexity is tackled by division into parts (lemmas versus subprograms and procedures).

According to Edsger, the programmer performs the following tasks. In the first two, elegance, accuracy, clarity, and a thorough understanding of the problem are prerequisite.

(1) Makes the complete specifications of the individual parts.

(2) Satisfies themself that the total problem is solved provided program parts meeting the various specifications are available.

(3) Constructs the individual parts to satisfy the specifications, but independent of one another and the further context in which they will be used.

Edsger says in EWD117 [Dijkstra 1965] that this dissection relies on what he calls "The principle of non-interference." By this he means that the working of the whole depends only on the specifications of the individual parts, and not on how the independent parts are implemented. He discusses language concepts and their implementation that adhere to this principle. The Algol 60 procedure adheres to this principle to a fair degree, he says, as does the implementation of arithmetic expressions, which requires local variables that the programmer doesn't know about. He does mention some other alternatives that turn out to have other disadvantages, and he ends up talking about the **goto** statement for perhaps the first time, asking himself "whether the **goto** statement as a remedy is not worse than the defect it aimed to cure." We'll come back to the principle of non-interference later in Section 7.8 on proving concurrent programs correct.

7.2.2 Three Mental Aids

Edsger wrote EWD227 [Dijkstra 1982b] in February 1968. In it, he outlines the mental aids he will use in discussing program construction, saying that "the mental aids available to the human programmer are, in fact, very few. They are enumeration, mathematical induction and abstraction." Edsger uses these mental aids in developing two quite different programs. Edsger also discussed these three mental aids in a talk in Grenoble in December 1967, but he didn't complete EWD241 [Dijkstra 1968e] on the topic until summer 1968.

Enumeration refers to the mental effort required to understand a code snippet like:

```
x := 5;
if x < 5 then begin y := 6; z := 25 * x end
```

but also to the snippet below, which appears in EWD241 as Edsger develops an algorithm:

```
if length of sequence equals 32 then
   begin accept sequence as solution;
         print solution
     end
```

The key points are: (1) The snippet is a fixed sequence of actions or conditional statements, (2) The number of cases is small enough for our small minds to grasp, and (3) Execution generates a small number of actions.

Edsger is not giving proof rules or anything like that; he is simply stating that if we keep the snippet short enough and the computations it invokes small enough, we can use normal mathematical-like reasoning to understand it.

Mathematical induction is the standard pattern of reasoning to understand loops and recursive procedures.

Abstraction is the main tool needed for the application of mathematical induction —as Edsger says, one has to "form the concepts in terms of which the net effect of the induction step can be described." For loops, these days we talk about the need for a loop invariant.

But abstraction is also needed to "reduce the appeal to enumeration." We see that above, where the then-part of the **if**-statement is given as *what* has to be performed (accept sequence as solution; print solution) and not *how* it is performed. Later, in EWD288 [Dijkstra 1970], Edsger called this *operational abstraction*, as opposed to *representational abstraction*.

This use of operational abstraction to separate the *what* from the *how*, which is mentioned in both this subsection and the previous one, is of utmost importance. Yet, today's programming courses rarely do justice to abstraction, and indeed it is hard to inspire students to use abstraction this way. Either they don't want to spend the time specifying a subroutine before writing its body or they find it too difficult. They waste much time and make many mistakes because of it.

Edsger devotes EWD264 [Dijkstra 1969] to showing the power of abstraction. Consider a sequence of n **if**–**else** statements, each with a specification S and with a specification for the overall sequence. Here's one **if**-statement with specification:

```
//S
if ... then ... else ...
```

For example, specification S: *Set* y *to the absolute value of* x can be implemented by: **if** $x < 0$ **then** $y := -x$ **else** $y := x$.

We count the "steps" involved in verifying the correctness of this sequence. Verifying that the sequence of specifications $S_0; S_1; \ldots; S_{n-1}$ satisfies the overall specification takes n steps, verifying that each S_i is implemented properly requires two steps (the if-condition is true or false), so verifying all of them requires $2n$ steps. Thus, in total, $3n$ steps are required.

On the other hand, suppose the **if**-statements are given without specifications:

```
if ... then ... else ...
...
if ... then ... else ...
```

Each if-condition can be true or false, so the sequence can be executed in 2^n ways. We have to verify them all. Verifying each will take n steps, so in total that's $n2^n$ verification steps!

Edsger showed so early in the game how important abstraction was; most people haven't listened.

7.3 Structured Programming, with Proof of Correctness the Main Concern

Early on, I can't say exactly when, Edsger began developing programs and their proofs hand-in-hand. That's the way he thought, that's the way he worked. A prime example of this is the development of an operating system, started in about 1962. In EWD196, published as Dijkstra [1968c], he writes:

> When the system has been delivered we shall not live in the perpetual fear that a system derailment may still occur in an unlikely situation such as might result from an unhappy "coincidence" of two or more critical occurrences, for we shall have proved the correctness of the system with a rigour and explicitness that is unusual for the great majority of mathematical proofs.

Later, in EWD209 [Dijkstra 1968a], completed in August 1967, Edsger develops —in stepwise fashion and with correctness the main concern— a concurrent program for the bounded buffer problem[1]. He writes,

> . . . the way in which it [the bounded buffer problem] is treated is representative for the way in which a whole multiprogramming system has actually

[1] Page 6 of EWD209 gives a simple description of his P and V operations on semaphores, which are used to construct critical sections and synchronize parallel processes. Edsger first discussed them in 1962 in EWD35 [Dijkstra 1962b]. Here is an English translation: EWD35-English [Dijkstra 1962c].

> been constructed. . . . We also publish it in the hope that it may serve as a partial answer to the many doubts evoked by our claim to have constructed a multiprogramming system of proven flawlessness.

Evidently, he received some flak about his earlier claims concerning the proof of an operating system. Further, he seemed to think that good programmers programmed this way only unconsciously. In one of the concluding remarks in EWD209 [Dijkstra 1968a], Edsger writes

> Firstly, one can remark that I have not done much more than to make explicit what the sure and competent programmer has already done for years, be it mostly intuitively and unconsciously. I admit so, but without any shame: making his behaviour conscious and explicit seems a relevant step in the process of transforming the Art of Programming into the Science of Programming. My point is that this reasoning can and should be done explicitly.

But EWD209 [Dijkstra 1968a] does not elaborate on the principles and strategies used to develop the bounded buffer program, and soon Edsger turned to doing just that. In EWD227 [Dijkstra 1982b], completed in February 1968, he develops two radically different algorithms: First, an algorithm to construct a table of the first 1,000 primes; second, an algorithm dealing with printing a page based on a paper tape created by a Flexowriter. In 1982, Edsger published a book containing about 80 of his EWDs [Dijkstra 1982a]. In introducing EWD227, Edsger says that this essay marked a turning point in his professional life: "... it represents my earliest conscious effort at orderly program development." EWD227 [Dijkstra 1982b] ends with this summary:

> Personally I am much more impressed by the similarity of the ways in which the two rather different programs have been constructed. The successive versions appear as successive levels of elaboration. It is apparently essential for each level to make a clear separation between "what it does" and "how it works". The description of "what it does", the definition of its net effect requires the introduction of the adequate concepts and both examples seem to show a way in which we can use our power of abstraction to reduce the appeal to be made upon enumeration.

Edsger worked on "orderly program development" in this style for the next two years or so, culminating in EWD288 [Dijkstra 1970], which appeared in July 1970, and of course his famous 82-page "Notes on structured programming" [Dijkstra 1972a], which he completed in April 1970 as EWD249. In the three-year period beginning August 1967, Edsger wrote over 25 EWDs that touched upon program development.

Let me try to distill the important points Edsger made concerning program development, with references where appropriate. I'll start with some general relevant quotations from his works.

(1) "[. . .] I have a very small head and had better learn to live with it to respect my limitations and give them full credit, rather than try to ignore then, for the latter vain effort will be punished by failure." [Dijkstra 1972a].

(2) "My thesis is, that a helpful programming methodology should be closely tied to correctness concerns. [. . .] If, however, he adheres to the discipline to produce the correctness proofs as he programs along, he will produce program and proof with less effort than programming alone would have taken." EWD288 [Dijkstra 1970].

(3) "[. . .] although the programmer only makes programs, the true subject matter of his trade are the possible computations evoked by them. [. . .] when a programmer claims that his program is correct, he actually makes a statement about the computations! [. . .] Trivial as this remark may seem I must state that it has had a profound influence on my thinking and my programming." (EWD245) [Dijkstra 1968f]. Thereafter, Edsger stuck to programming constructs that made it easiest to understand the dynamic computations from the static program.

(4) "[. . .] program correctness is not my only concern, program adaptability or manageability will be another." "My refusal to regard efficiency considerations as the programmer's prime concern is not meant to imply that I disregard them. [. . .] we can only afford to optimize [. . .] provided the program remains sufficiently manageable." [Dijkstra 1972a].

(5) Separation of concerns: "This is what I mean by 'focusing one's attention upon some aspect': it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant." Edsger coined the term in EWD447, "On the role of scientific thought" [Dijkstra 1982c], completed in August 30, 1974, giving it a far broader meaning that just in programming. We use it, for example, to focus attention on one aspect at a time, for example, correctness versus efficiency, and at a lower scale, which step to take next when developing an algorithm by stepwise refinement.

(6) "I would not dare to suggest (at least at present!) that it is the programmer's duty to supply a proof whenever he writes a simple loop [. . .] when

> a programmer considers a construction like (6) [a simple loop] as obviously correct, he can do so because he is familiar with the construction. I prefer to regard his behaviour as an unconscious appeal to a theorem he knows [. . .]." [Dijkstra 1972a].

Let me now list major points concerning stepwise refinement, as espoused by Edsger. I hope this brief glimpse into Edsger's thinking will inspire you to read the cited EWDs in order to gain more insight into his thinking and to see examples of Edsger performing stepwise refinement.

(1) Use Enumeration, Mathematical induction, and Abstraction to prove programs correct, as discussed in subsection 7.2.1.

(2) Reduce the amount of Enumeration needed by *operational abstraction*, as discussed in subsection 7.2.2. In developing a program, one step might be to replace an English statement by a sequence of other statements, written in terms of *what* each does. Later refinement steps implement those statements, defining *how* they are carried out.

(3) *Representational abstraction* is concerned with compound data structures. At one point of development, we may introduce a variable to contain a set of values of some type; this indicates *what* is needed. Much later, we decide on an implementation of that set, indicating *how* those values are represented. Such a refinement may call for a refinement of statements that refer to that set of values (EWD288 [Dijkstra 1970]).

(4) *On program families*: "In this section, I am going to explain why I prefer to regard a program not so much as an isolated object, but rather as a member of a family of 'related programs'." They are related, "either as alternative programs for the same task or as similar programs for similar tasks" [Dijkstra 1972a, "Notes on structured programming", p. 50]

Edsger devotes three pages to this topic. He ends with this: "[. . .] program structure should be such as to anticipate its adaptations and modification. Our program should not only reflect (by structure) our understanding of it, but it should also be clear from its structure what sort of adaptations can be catered for smoothly [. . .]."

7.4 1968 NATO Conference on Software Engineering

Let me now discuss a turning point in the history of software development and Edsger's involvement in it. In October 1968, the landmark NATO Conference on Software Engineering took place [Naur and Randell 1969]. The conference introduced the phrase *software engineering*, and people from both academia and industry admitted to their problems in developing software and hardware. The conference was unique in that all discussions were recorded, and after the conference, many were transcribed and placed in the final report. If you are interested in learning about this period of computing, read the report. A link to it can be found in Naur and Randell [1969].

In preparation for the NATO conference, in about June 1968, Edsger completed EWD236 [Dijkstra 1968d]. In it, Edsger reviews the experiences gained during the design and construction of the THE Multiprogramming System. He writes, "The difficulties that have been overcome reasonably well are related to the reliability and the producibility of the system, the unsolved problems are related to the sequencing of the decisions in the design process itself." He goes on to describe the operating system as a sequence of machines: $A[0], A[1], \ldots, A[n]$, with $A[0]$ being the given hardware machine, and the software of layer i transforms machine $A[i]$ into $A[i+1]$.

Session 4.3.2 of the conference, titled "Structuring the design," began with presentations of two working papers —Edsger's EWD236 [Dijkstra 1968d] and one by Brian Randell— followed by a discussion. Also, during the conference, Edsger made many comments about the design process. Here is one:

> [. . .] However, I am convinced that the quality of the product can never be established afterwards. Whether the correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made. This means that the ability to convince users, or yourself, that the product is good, is closely intertwined with the design process itself.

Edsger also explains why people don't produce the necessary documentation at the appropriate moment:

> [. . .] making the predocumentation at the proper moment, and using it, will improve the efficiency with which you construct your whole thing incredibly. One may wonder, if this is so obvious, why doesn't it happen? I would suggest that the reason why many programmers experience the making of predocumentation as an additional burden, instead of a tool, is that whatever predocumentation he produces can never be used mechanically.

The reason, of course, is that that predocumentation has to be written in a mixture of natural language and mathematics.

What did Edsger think of this conference? Here's what he wrote two years later, in EWD288 [Dijkstra 1970]:[2]

> For me, the Conference [. . .] has been a turning-point: it was the first time I witnessed a group of experts —all of them so high in their local hierarchies that they could afford to be honest— unanimously admitting that a software crisis did indeed exist. This struck me as very important, because before its existence was admitted it was vain to hope that something could be done about it.

But not too long after the conference, he typed a two-page trip report on it, EWD246 [Dijkstra 1968g]. For the first time, he wrote, it was difficult for him to write a report after a trip. He had heard so much that he hadn't yet sorted it out. He did find the conference exciting because he knew only about half the people, and he appreciated the candidness of almost all the participants in discussing their problems with developing hardware and software. He had some harsh words for the IBM 360 operating system. And he found it striking what Americans said about the place of Software Engineering in Education. It couldn't be entrusted to Math Departments ("the real mathematician is utterly useless," he wrote) and certainly not to Departments of Computer Science. Edsger wrote that, "To refresh the discussion I mentioned 'The mathematical engineer,' but many experienced this as a contradiction in terms." And he states that, "the software failure is now so manifest that it is no longer possible to turn a blind eye to it." EWD246 [Dijkstra 1968g] ends with the following paragraph (translated to English) —the project on which he is now working is the development of a helpful programming methodology with correctness being the main concern:

> As a person, I have been listening very closely to gauge the significance of the project on which I am now working. I am more than ever convinced of its importance [. . .] Overall, the experience is an incentive to continue on the chosen path; I kept getting the impression that by now I had plowed deeper than others. [. . .] In short, my constant effort to turn my weakness into my strength has only been reinforced if possible throughout the confrontation. So far my report, perhaps only in the first round.

I was at this NATO conference. I was 29 years old, with little experience in computing research; Edsger was 9 years older and had been doing groundbreaking work in computing for over 15 years. Two years before, I had received my Ph.D.

[2] Dijkstra put a space before and after each parenthetical remark delimited by em dashes —as in this one— because without those spaces, it was not always easy to see whether the em dash began or ended the parenthetical remark. This style has been adopted in all quotes from his works.

(actually, Dr. Rer. Nat.) from Fritz Bauer in Munich, the main organizer of this conference, and in 1966 I began teaching in the newly founded CS Department at Stanford. I had been the main implementor of an Algol 60 compiler [Gries et al. 1965], translating the detailed instructions that the designers of the compiler, Manfred Paul and Hans Ruediger Wiehle, gave me into IBM 7090 assembly language. We began this work at the University of Illinois in the US, and I finished it in Munich, Germany. But my thesis was in mathematics. I did not know Edsger (and most of the other people at the conference), and he and I did not exchange a word, although I have a photo taken during a break where we are rather close together.

7.5 1968–1969: A Period of Suffering

It's remarkable that Edsger did all this momentous work on program-proof development at a time when he was quite distressed and depressed. An inkling of his mental state is given on the first page of EWD227 [Dijkstra 1982b], completed in February 1968, where he writes,

> [. . .] it is the kind of peaceful prose that I write (mainly for my own distraction?) when a somewhat poor condition forces me for some period of time to some sort of inactivity. It will certainly be less gloomy than this evening's front page news!

He clarified this somewhat in Dijkstra [1982a], a collection of over 60 of his EWDs. In it, he refers to EWD227 [Dijkstra 1982b] in a dramatic way:

> This essay, though dating from February 1968, has been included because, in retrospect, it marks a turning point in my professional life: it represents my earliest conscious effort at orderly program development. The whole essay —and this explains to a certain extent its somewhat pathetic covering letter— was written while I felt mortally ill: it was written as my farewell to science. Fate has decided differently.

(I edited this collection of EWDs [Dijkstra 1982a] for him, but at the time I did not catch on to this admission of his state of mind.) It was a hard time for Edsger, but he pursued what he felt was his calling. Few scientists have injected personal information like this in their writings.

About 23 years later, in June 2001, Edsger wrote the following in EWD1308 [Dijkstra 2002], "What led to 'Notes on structured programming'":

> In 1968 I suffered from a deep depression, partly caused by the Department, which did not accept Informatics as relevant to its calling and disbanded the group I had built up, and partly caused by my own hesitation what to do next.

> I knew that in retrospect, the ALGOL implementation and the THE Multiprogramming System had only been agility exercises and that now I had to tackle the real problem of How to Do Difficult Things. In my depressed state it took me months to gather the courage to write (for therapeutic reasons) EWD249 "Notes on Structured Programming" (August 1969); it marked the beginning of my recovery.

Sadly, the Department of Mathematics at the Eindhoven University of Technology did not appreciate his work and disbanded his group. Yet, he had the courage to plow on in what he felt was important work, and it was therapeutic for him. Later, in 1973, he began working from home as a research fellow at the Burroughs Corporation, which helped him to distance himself from the source of his frustration. He did continue to maintain a connection to the University, going in once a week.

One other point strikes me. For Edsger, the THE Multiprogramming System and the Algol implementation were only agility exercises! Yet, they were unique. Who else has written an operating system that was proven correct (except for trivial typos)? Who else wrote the first ALGOL 60 compiler, with all sorts of new inventions in compiling to make it work?

7.6 The Development of Weakest Preconditions

We now turn to Edsger's development of weakest preconditions and a calculus for the derivation of programs, in which, at times, program snippets can be *calculated* instead of guessed. Taking this step was not easy for Edsger, as he admits in EWD1308 [Dijkstra 2002]:

> Looking back I cannot fail to observe my fear of formal mathematics at the time. In 1970 I had spent more than a decade hoping and then arguing that programming would and should become a mathematical activity; I had (re)arranged the programming task so as to make it better amenable to mathematical treatment, but carefully avoided creating the required mathematics myself. I had to wait for Bob [Floyd 1967], who laid the foundation, for Jim King, who showed me the first example that convinced me, and for Tony Hoare [Hoare 1969], who showed how semantics could be defined in terms of the axioms needed for the proofs of properties of programs, and even then I did not see the significance of their work immediately. I was really slow.

Edsger first mentions weakest preconditions in EWD360 [Dijkstra 1973a], which was completed in Spring 1973. It's 22 typed pages. In order to describe Edsger's approach to weakest preconditions and logic, we have to compare it with Tony

Hoare's approach to defining the semantics of a small programming language in his seminal paper [Hoare 1969]. Tony, the consummate logician, wrote,

> Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs.

Tony introduced the notation $\{P\}\, S\, \{R\}$, which is now called a *Hoare triple*, with meaning: If execution of statement S in a state in which predicate P is true terminates, R will be true.[3]

Here's Tony's axiom of assignment:

$$\vdash \{R^x_e\}\; x := e\{R\}, \tag{7.1}$$

where R^x_e denotes R with every occurrence of expression e replaced by variable x. Here's Tony's Composition inference rule:

$$\text{If } \vdash \{P\}\, S1\, \{Q\} \text{ and } \vdash \{Q\}\, S2\, \{R\} \text{ then } \vdash \{P\}\, (S1; S2)\, \{R\}. \tag{7.2}$$

In EWD360 [Dijkstra 1973a], Edsger treats this differently. Instead of putting precondition Q and postcondition P on an equal footing as in the Hoare triple $\{P\}\, S\, \{R\}$, he defines one in terms of the other. He defines the weakest precondition $wp(S, R)$ to be a predicate that describes the set of all states in which execution of S is guaranteed to terminate in a state in which R is true.[4]

The connection between the Hoare triple and the weakest precondition is this: if $P \Rightarrow wp(S, R)$, then $\{P\}\, S\, \{R\}$. The converse implication does not hold since $\{P\}\, S\, \{R\}$ does not ensure termination of all executions of S that start in a state in which predicate P is true.[5]

[3] Tony actually wrote $P\{S\}\, R$, but this notation was soon changed to $\{P\}\, S\, \{R\}$, which also allows us to insert assertions into the text, by just putting $\{P\}$ at an appropriate place. This convention became particularly useful for the definition of interference-freedom, discussed in Section 7.8.

[4] In EWD360, Edsger used the notation $fS(R)$ for the weakest precondition. He later switched to $wp(S, R)$. The Dijkstra–Scholten book [Dijkstra and Scholten 1990] uses the notation $wp.S.R$.

[5] To ensure equivalence Edsger introduced the *weakest liberal precondition, wlp*. $wlp(S, R)$ is a predicate that describes the set of all states from which all terminating executions of S end in a state in which R is true. The Dijkstra–Scholten book [Dijkstra and Scholten 1990] discusses *wlp* extensively; I don't know the first reference to *wlp* in Edsger's EWDs.

This use of predicate transformers instead of Hoare triples then allows Edsger to derive programs in stepwise fashion hand-in-hand with their *formal* proofs —and even to *calculate* parts of programs instead of just guessing. We'll study this topic in a moment.

But Edsger changed something else: He got rid of logic, as Tony used it. He rewrote Tony's axiom for assignment and inference rule Composition:

$$\text{Axiom of Assignment: } wp(\text{“}x := e\text{”}, R) = R^x_e \tag{7.3}$$

$$\text{Axiom of Composition: } wp(\text{“}S1; S2\text{”}, R) = wp(S1, wp(S2(R))) \tag{7.4}$$

So, the Composition Inference rule becomes an axiom. Instead of logic, he uses an algebraic calculational system. This is what he wrote to me much later, in 1995, in EWD1227 [Dijkstra 1995b], a somewhat open letter to me: "... an algebra is not 'explained', it is 'postulated'! One postulates a domain with a few operators and relations with certain properties, and usually there are a few existence axioms as well, and the result can be viewed as a calculational system."

What is the difference between a predicate logic and the corresponding predicate algebra? Rutger Dijkstra, Edsger's son and a mathematician and computer scientist in his own right, explained his view of this much later in a paper titled "'Everywhere' in predicate algebra and modal logic" [Dijkstra 1996], which I had urged him to write. He says that in a predicate logic a predicate is an expression, while in a predicate algebra a predicate is the value of an expression. For example, $(P \equiv Q) \equiv R$ and $P \equiv (Q \equiv R)$ are different predicates in predicate logic but the same predicate in predicate algebra because operator $\equiv$ is associative (so they have the same value). We discuss this topic briefly in Section 7.7.

Let's return to Edsger's development of weakest preconditions and a calculus for the derivation of programs. He's going full force on this, writing next EWD367 [Dijkstra 1973b], completed on April 29, 1973, which, he says, "is heavily influenced by the work of C.A.R. Hoare." It's 17 typed pages. In it he lays out the properties that predicate transformers wp must have, such as $Q = R$ implies $wp(S, Q) = wp(S, R)$ and the Law of the Excluded Middle, $wp(S, F) = F$. He develops theorems concerning (recursive) procedures and the **while** loop. He then presented this work at a meeting in Munich of IFIP Working Group 2.3. Eleven days later, he revised this work based on comments given at the meeting and comments from his colleagues in Eindhoven. The result was EWD372 [Dijkstra 1973c].

IFIP Working Group 2.3, on Programming Methodology, was a great place to present what one was working on and get feedback from others. Most speakers presented not completed published work but what they were working on and asked for suggestions and advice. We need more venues like this.

Six months later, Edsger completed EWD398 [Dijkstra 1973d], in which he introduced the guarded command **if**-statement and repetition[6]:

$$
\begin{array}{ll}
\textbf{if } B_0 \longrightarrow S_0 & \qquad \textbf{do } B_0 \longrightarrow S_0 \\
\ [\,]\ B_1 \longrightarrow S_1 & \qquad \ [\,]\ B_1 \longrightarrow S_1 \\
\ \ldots & \qquad \ \ldots \\
\textbf{fi} & \qquad \textbf{od}
\end{array}
\tag{7.5}
$$

This is significant because it introduces explicit nondeterminism into a programming language for the first time. For example, the guarded command **if**-statement is executed as follows: If one (or more) condition B_i is true, arbitrarily choose one that is true and execute the corresponding command S_i. Otherwise —if no condition B_i is true— abort. Naturally, these guarded command statements are defined in terms of weakest preconditions. Their use actually smoothens some stepwise refinements.

The following statement to store the maximum of x and y in z shows nondeterminacy; if $x = y$, either branch may be executed.

$$
\begin{array}{l}
\textbf{if } \ x \geq y \longrightarrow z := x \\
\ [\,]\ y \geq x \longrightarrow z := y \\
\textbf{fi}
\end{array}
$$

As shown by Edsger, this little example can be partially calculated, as follows. The postcondition is $R : (z = x \wedge x \geq y) \vee (z = y \wedge y \geq x)$. An obvious statement to truthify postcondition R is statement $S : z := x$. One can *calculate* $wp(S, R)$ and manipulate the result to arrive at the guard $x \geq y$ and thus end up with the guarded command $x \geq y \longrightarrow z := x$.

One large problem concerned loops: A loop invariant is needed to establish correctness of a loop. Edsger developed several ways to generalize the precondition and postcondition to arrive at a loop invariant.

This research ended up first with the article Dijkstra [1975] and then the research monograph *A Discipline of Programming* [Dijkstra 1976]. It took only three years from the start of this research to the publication of this book.

But that's not the end of this story. Throughout the 1980s, 1990s, and beyond, there was much discussion among the community about the formal development of various algorithms. Textbooks were written on the topic, the first being

[6] The notation we use here is his final one, and not the one original used in EWD398 [Dijkstra 1973c].

my own *The Science of Programming*, in 1981 [Gries 1981], for which Edsger wrote the foreword. Edsger's monograph was structured to some extent on algorithms. In formally developing an algorithm, he discussed the strategies and techniques being used. My text, which he and I often called "Dijkstra for the Masses," turned it around: It discussed each strategy and technique in turn, giving examples of its use.

Articles appeared that explained and extended the developmental strategies, for example, my own article [Gries 1982]. Even in 1999, Rudolph Berghammer and Thorsten Hoffman published an article that combined relation calculus with Edsger's method for deriving programs [Berghammer 1999].

More texts were also published, for example, *Program Construction and Verification* [Backhouse 1986], *Programming: The Derivation of Algorithms* [Kaldewaij 1990], *Programming from Specifications* [Morgan 1994], and *Refinement Calculus: A Systematic Introduction* [Back and von Wright 2008].

Today, in a course on data structures, with a suitable introduction to proofs of correctness and the development of loop invariants, we can teach several sorting algorithms in one lecture —insertion sort, selection sort, and quicksort— not by presenting them as completed products but by *developing* them. Further, students can now know these algorithms because they can develop them whenever they need to.

All this is due to Edsger.

7.7 A Calculus Based on Leibniz's Substitution of Equals for Equals

Edsger treated his weakest precondition transformer $wp(S, R)$ as part of an algebra, a calculational system that uses logical operators $\neg, \vee, \wedge, \equiv$ and $\Rightarrow$. Because Leibniz's rule of substitution (shown below) played a major part in his formal calculations, he was investigating properties of equivalence $\equiv$. EWD842 [Dijkstra 1982e] records Edsger's first recognition that the associativity of equivalence $\equiv$ could provide a significant economy of notation. It is not a finished product, ready for publication; it simply records a revelation that came to him and its consequences. I can almost hear him chuckling as he wrote EWD842. At the end of it he writes, "Since I don't want to become a logician I had better stop; in any case I have had my fun."

Edsger knew that logician Jan Łukasiewicz [1970] knew that $\equiv$ was associative as far back as 1918, but for Łukasiewicz and other logicians it was not significant. Logicians didn't *use* logic that much, they studied it. Gerhard Gentzen [1934] did develop a logic called *Natural Deduction* in order "to set up a formal system that comes as close as possible to natural reasoning," but it relied on *modus ponens* and not Leibniz's rule of substitution, and associativity of $\equiv$ played no part. Much later, in 1981, Edsger wrote in EWD803 [Dijkstra 1981] that equality of propositions is a curiously neglected concept and suggests that is because our Western languages are not very well suited for its verbal expression. He goes on to say:

> A major purpose of formalisms is to liberate who masters them by freeing him —and his thinking— from the shackles he inherited when he learned his native tongue. (Gentzen's system of so-called "natural deduction" was aimed at formalizing how arguments are expressed in natural languages, and that is precisely why I consider it a great step backwards: it fails to liberate.)

Rather than discuss EWD842 [Dijkstra 1982e] in detail —I leave the enjoyment of reading it to reader— let me give Leibniz's rule and show how use of associativity (and also symmetry) of $\equiv$ provides such an economy of notation.

Here is Leibniz's rule, essentially as Edsger expressed it in EWD842:

> Leibniz's Rule: In the presence of theorem $P \equiv R$, a new theorem may be formed by replacing in an existing theorem one or more occurrences of P by R.

Now, for many, $x = y = z$ is shorthand for $x = y \wedge y = z$. By extension, since $\equiv$ is operator $=$ restricted to Booleans, $P \equiv Q \equiv R$ is shorthand for $P \equiv Q \wedge Q \equiv R$. But we no longer use this convention for $\equiv$. Instead, as Edsger says in EWD842, it can be parsed in two ways:

$$P \equiv Q \equiv R \text{ stands for either (or both) } P \equiv (Q \equiv R) \text{ or } (P \equiv Q) \equiv R,$$

A major economy of notation occurs in what Edsger much later[7] called the *Golden rule*:

$$\text{Golden rule: } P \wedge Q \equiv P \equiv Q \equiv P \vee Q. \tag{7.6}$$

Using associativity, the Golden rule can be parsed in four different ways, but if one also allows symmetry of $\equiv$, there are many more. Below are two of them. The first is an axiom that introduces operator $\wedge$ for the first time in Edsger's algebra. It is the definition of $\wedge$. The second is a theorem that is well known to many. The economy of notation allowed by associativity and symmetry of $\equiv$ is indeed mind boggling.

$$P \wedge Q \equiv (P \equiv Q \equiv P \vee Q) \tag{7.7}$$

$$(P \equiv Q) \equiv (P \wedge Q \equiv P \vee Q). \tag{7.8}$$

[7] The first occurrence of the Golden rule that I found was in EWD863 [Dijkstra 1983], "Predicate calculus revisited," completed October 13, 1983.

This calculational system is the topic of the Dijkstra–Scholten book *Predicate Calculus and Program Semantics* [Dijkstra and Scholten 1990]. The Gries–Schneider text *A Logical Approach to Discrete Math* [Gries and Schneider 1993] uses a similar predicate logic to teach discrete math. An advantage of this logic is that one can introduce and teach principles and strategies for developing proofs. For those who are not familiar with the calculational approach, we give a proof of the theorem $p \vee true \ \equiv \ true$, using the style developed by Edsger and his colleagues. Each line starting with $=$ has an indented part, which indicates the rule used to show the equality (or equivalence) using Leibniz's rule of substitution.

$$\begin{array}{ll}
 & p \vee true \\
= & \quad < \text{Identity of } \equiv > \\
 & p \vee (p \equiv p) \\
= & \quad < \text{Distributivity of } \vee \text{ over } \equiv > . \\
 & (p \vee p) \equiv (p \vee p) \\
= & \quad < \text{Identity of } \equiv > \\
 & true
\end{array}$$

The Everywhere Operator

At some point, Edsger began using the notation $[P]$, with this meaning: $[P]$ is true if predicate P is true in all states and false otherwise. $[P]$ is called the *everywhere operator*. The first use of it that I could find is EWD813 [Dijkstra and Scholten 1982], dated March 17, 1982, and written with C.S. Scholten. EWD813 doesn't define $[P]$, but it does contain this statement: "$[P = Q]$ is a boolean —true if and only if in each state P and Q have equal values." The purpose of EWD813 is to investigate the relation between predicate transformers *wp* and *wlp*. Along the way, the monotonicity of predicate transformers f is introduced:

$$[P \Rightarrow Q] \Rightarrow [fP \Rightarrow fQ] \quad \text{for all } P \text{ and } Q. \tag{7.9}$$

The everywhere operator is used in many later EWDs, especially those that are drafts or actual chapters of the final book by Edsger and Carel Scholten, *Predicate Calculus and Program Semantics* [Dijkstra and Scholten 1990], and it plays an important role in this book.

I must admit that I did not understand the everywhere operator at the time. I proofread [Dijkstra and Scholten 1990] for him carefully, but when it came to the everywhere operator, I was confused. Instead of telling Edsger this, I simply did nothing. It was a mistake, and I apologized much later. Therefore, when

Fred Schneider and I were writing *A Logical Approach to Discrete Math* [Gries and Schneider 1993] —at its core is a logic, not an algebra—, we used Edsger's approach to axioms for the logical operators but did not include the everywhere operator.

Others had issues with the use of the everywhere operator in Dijkstra and Scholten [1990], and there were negative reviews of this book. One colleague who knew Edsger well wrote me that, although he used the everywhere operator, he did not understand it fully.

Then, as Fred and I struggled to come to grips with the everywhere operator, we had correspondences with Edsger and also his son Rutger Dijkstra. This resulted in three articles:

(1) In EWD1227 [Dijkstra 1995b], Edsger says, "You seem to doubt Rutger's statement '. . . when it comes to being short, simple, illuminating, and convincing, algebraic proofs and logical deductions are simply not in the same league.'" Well I still doubt that as it pertains to predicate algebra versus predicate logic, as we see later. But he is right in other ways, which he explains in EWD1227. Here's one: "Another advantage of algebra over logic is the ease with which one can define new algebras from old, for example by forming direct products and function spaces." I wrote a reply, "A somewhat open letter to Edsger W. Dijkstra," which was probably read only by him.

(2) Rutger's article [Dijkstra 1996], which I encouraged him to write after several conversations. In it, Rutger defines the term "predicate algebra" and presents the predicate algebra of Dijkstra and Scholten [1990]. He presents the propositional part of Edsger's predicate algebra together with the everywhere operator [. . .] and indicates that it is just a version of modal logic S5. Thus, $[P]$ is the well-known modal operator $\Box P$. He says at one point that the language used in Dijkstra and Scholten [1990] is very similar to the language that is the object of study in logic. "This, together with the fact that they recycled some vocabulary, appears to have caused some misunderstanding."

(3) The article [Gries and Schneider 1998]. Based on our correspondences with Rutger, we were able to develop a calculational logic (using at its core Leibniz's substitution of equals for equals) for propositional logic augmented with $\Box P$. As a follow-up, Warford, Vega, and Staley [Warford et al. 2020] developed a calculational deductive system for linear temporal logic, in which all proofs are given simply in the calculational style.

The everywhere operator is also mentioned in Chapter 11 written by Vladimir Lifschitz. He discusses its use in the context of the calculation proofs of Dijkstra and Scholten.

7.8 Proofs of Concurrent Programs

Earlier, we mentioned Edsger's "principle of non-interference": the working of the whole should depend only on the specifications of the individual parts and not on how the independent parts are implemented. He discussed this early in 1965 in EWD117 [Dijkstra 1965].

Consider two processes *S*1 and *S*2, which share a few variables. Each has been proved correct on its own. But when they are executed in parallel, correctness goes out the window because of the shared variables.

In her Ph.D. thesis of 1975 [Owicki 1975], my student Susan Owicki developed the notion of interference freedom for concurrent programs. It defined what else has to be established to ensure that concurrent execution of *S*1 and *S*2 is correct. This led to articles Owicki and Gries [1976a] and Owicki and Gries [1976b]. I was fortunate in August 1975 to attend the NATO Summer School in Marktoberdorf, Germany. There, Edsger devoted three lectures to a concurrent problem, on-the-fly garbage collection. I saw immediately that Susan's interference freedom technique could be used here, spent some time working on it, and presented it in a one-hour lecture at the end of the School.

Edsger read Susan's thesis and, in March 1976, completed EWD554, later published as Dijkstra [1982d]. In it, he calls Owicki's work the first significant step toward applying axiomatic methods to concurrent processing. (Leslie Lamport also independently conceived of the idea of interference-freedom, but in the setting of concurrent programs presented in the form of parallel flowcharts [Lamport 1977].)

However, none of this work dealt with the problem of how to *develop* concurrent programs hand-in-hand with their correctness proofs. I certainly didn't think of this issue at the time, for work on the development of *sequential* programs was just in its infancy. Who could think about development of concurrent ones? Edsger's article [Dijkstra 1975] was just published the same month, August 1975, and the field spent the next 10–15 years learning about, extending, and writing about the development of sequential programs.

The first book to address the development of concurrent programs appeared 22 years later. It was by my friend and colleague at Cornell, Fred Schneider, titled *On Concurrent Programming* [Schneider 1997]. Fred says in Chapter 27 that just as Edsger used loop invariants to calculate the guard that terminates a loop and to calculate assignment statements in the repetend, he would use the interference-freedom obligation to motivate steps in the derivation of a concurrent program. Read Fred's Chapter 27 to see the influence Edsger had on Fred.

Two years later, Edsger's close collaborators Wim Feijen and Netty van Gasteren [1999] published *On a Method of Multiprogramming*. It is based on the Owicki–Gries theory but it also emphasizes principles espoused time and again by Edsger, such as *stepwise refinement* in small steps, *separation of concerns*, and *the need for fluency in predicate calculus* if one is going to calculate formally. Edsger's positive influence is seen throughout this book.

7.9 Conclusion

In the mid-to-late 1960s, few programmers thought about how to reason about program correctness. They just wrote their programs and tried to debug them. Research on reasoning about correctness was just beginning, for example with the work of Tony Hoare and Peter Naur.

But in this period, Edsger Dijkstra was already thinking and writing about the *development process*: how to construct a program and its proof hand-in-hand, with the proof ideas leading the way. First with informal proofs and later with formal proofs, with some parts of a program actually being calculated.

His work on *program development* had a profound influence on those around him, including those to whom he sent his EWDs through the mail and later through email.

But over the years, this was not a one-man show at all. Edsger didn't work in a vacuum. He, like us and perhaps even more, needed support and constructive criticism. Many of us can recall sitting with him, carefully reading and discussing something that he, we, or someone else had written. His "Tuesday Afternoon Club," in which a bunch of people would read an article line by line and critique each word and sentence, have become widely known.

And, of course, many others have contributed new ideas, new research, and extensions of the work on programming methodology, as well as contributions to education in this field.

Still, Edsger was the heart of it all.

Over the years, Edsger shifted his interests from programming methodology to mathematical methodology, and much of his later research life was devoted to the latter. Here's a quote from EWD1209 [Dijkstra 1995a]:

> As a matter of fact, the challenges of designing high-quality programs and of designing high-quality proofs are very similar, so similar that I am no longer able to distinguish between the two: I see no meaningful difference between programming methodology and mathematical methodology in general. The long and short of it is that the computer's ubiquity has made the ability to apply mathematical method[s] more important than ever.

I find it sad that mathematicians, by and large, have not adopted Edsger's ideas on mathematical methodology.

There is another way of looking at Edsger's contribution to computing. He, more than anyone else, continually emphasized the need for style and what some might feel are vague goals like beauty, simplicity, elegance, and clarity. He wrote EWD619 [Dijkstra 1977], "Essays on the nature and role of mathematical elegance," in order "Firstly, to show what effective thinking is all about, secondly, to show the circumstances under which it is indispensable, and thirdly, to try to convince readers that it can be taught." He goes on to say that,

> in (the practice of!) automatic computing, elegance is not a dispensable luxury, but a must, a "matter of life and death" so to speak.

And, in EWD697, "Some beautiful arguments using mathematical induction" [Dijkstra 1980], he writes,

> [. . .] when we recognize the battle against chaos, mess, and unmastered complexity as one of computing science's major callings, we must admit that "Beauty is our Business."

At least for me, this continual stress on beauty, simplicity, elegance, and clarity struck a chord, and I too now strive in my writings to fully embrace these qualities.

Thus, when it came time to celebrate his 60th birthday and sixty-one colleagues wrote technical contributions for the collection of articles, the natural title for the book was *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra* [Feijen et al. 1990].

References

R.-J. Back and J. von Wright. 2008. *Refinement Calculus: A Systematic Introduction*. Springer, New York.

R. C. Backhouse. 1986. *Program Construction and Verification*. Prentice-Hall International, Englewood Cliffs, NJ.

R. Berghammer. 1999. Combining relational calculus and the Dijkstra–Gries method for deriving relational programs. *Inf. Sci.* 119, 3–4, 155–171. DOI: https://doi.org/10.1016/S0020-0255(99)00012-2.

E. W. Dijkstra. 1962a. Some meditations on advanced programming. In *IFIP Congress*. North-Holland, 535–538.

E. W. Dijkstra. 1962b. Over de sequentialiteit van procesbeschrijvingen. EWD35. https://www.cs.utexas.edu/∼EWD/ewd00xx/EWD35.PDF.

E. W. Dijkstra. 1962c. About the sequentiality of process descriptions. EWD35 in English. https://www.cs.utexas.edu/∼EWD/translations/EWD35-English.html.

E. W. Dijkstra. 1965. Programming considered as a human activity. EWD117. https://www.cs.utexas.edu/∼EWD/ewd01xx/EWD117.PDF.

E. W. Dijkstra. 1968a. A constructive approach to the problem of program correctness. *BIT Numer. Math.* 8, 174–186. DOI: http://doi.org/10.1007/BF01933419.

E. W. Dijkstra. 1968b. Go to statement considered harmful. *Commun. ACM* 11, 3, 147–148. DOI: https://doi.org/10.1145/362929.362947.

E. W. Dijkstra. 1968c. The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5, 341–346. DOI: https://doi.org/10.1145/363095.363143.

E. W. Dijkstra. 1968d. Complexity controlled by hierarchical ordering of function and variability. EWD236. https://www.cs.utexas.edu/∼EWD/ewd02xx/EWD236.PDF.

E. W. Dijkstra. 1968e. Towards correct programs. EWD241. https://www.cs.utexas.edu/∼EWD/ewd02xx/EWD241.PDF.

E. W. Dijkstra. 1968f. On useful structuring. EWD245. https://www.cs.utexas.edu/∼EWD/ewd02xx/EWD245.PDF.

E. W. Dijkstra. 1968g. Verslag van het bezoek aan de NATO Conference on Software Engineering. EWD246. https://www.cs.utexas.edu/∼EWD/ewd02xx/EWD246.PDF.

E. W. Dijkstra. 1969. On understanding programs. EWD264. https://www.cs.utexas.edu/∼EWD/ewd02xx/EWD264.PDF.

E. W. Dijkstra. 1970. Concern for correctness as a guiding principle for program construction. EWD288. https://www.cs.utexas.edu/∼EWD/ewd02xx/EWD288.PDF.

E. W. Dijkstra. 1972a. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.), *Structured Programming*. Academic Press Ltd, 1–82. Originally published as EWD249 in April 1970.

E. W. Dijkstra. 1972b. Advanced course on computer systems architecture. EWD356. https://www.cs.utexas.edu/∼EWD/ewd03xx/EWD356.pdf.

E. W. Dijkstra. 1973a. On the necessity of correctness proofs. EWD360. https://www.cs.utexas.edu/∼EWD/ewd03xx/EWD360.pdf.

E. W. Dijkstra. 1973b. On the axiomatic definition of semantics. EWD367. https://www.cs.utexas.edu/∼EWD/ewd03xx/EWD367.PDF.

E. W. Dijkstra. 1973c. A simple axiomatic basis for programming language constructs. EWD372. https://www.cs.utexas.edu/∼EWD/ewd03xx/EWD372.PDF.

E. W. Dijkstra. 1973d. Sequencing primitives revisited. EWD398. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD398.PDF.

E. W. Dijkstra. 1975. Guarded commands, nondeterminacy, and formal derivations of programs. *Commun. ACM* 18, 8, 453–457. DOI: https://doi.org/10.1145/360933.360975.

E. W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall. ISBN 013215871X.

E. W. Dijkstra. 1977. Essays on the nature and role of mathematical elegance. EWD619. https://www.cs.utexas.edu/∼EWD/ewd06xx/EWD619.PDF.

E. W. Dijkstra. 1980. Some beautiful arguments using mathematical induction. *Acta Inform.* 13, 1, 1–8. ISSN 0001-5903 (print), 1432-0525 (electronic). DOI: https://doi.org/10.1007/BF00288531.

E. W. Dijkstra. 1981. Lecture notes on the structure of programs and proofs. EWD803. https://www.cs.utexas.edu/∼EWD/ewd08xx/EWD803.PDF.

E. W. Dijkstra. 1982a. *Selected Writings on Computing: A Personal Perspective*. Springer Verlag, New York.

E. W. Dijkstra. 1982b. EWD227: Stepwise program construction. In E. W. Dijkstra (Ed.), *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, New York, 1–12. Completed February 1968.

E. W. Dijkstra. 1982c. EWD447: On the role of scientific thought. In E. W. Dijkstra (Ed.), *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, New York, 60–65. Completed August 1974.

E. W. Dijkstra. 1982d. EWD554: A personal summary of the Gries-Owicki theory. In E. W. Dijkstra (Ed.), *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, New York, 188–199. Completed March 1976.

E. W. Dijkstra. 1982e. Two cheers for equivalence. EWD842. https://www.cs.utexas.edu/∼EWD/ewd08xx/EWD842.PDF.

E. W. Dijkstra. 1983. Predicate calculus revisited. EWD863. https://www.cs.utexas.edu/∼EWD/ewd08xx/EWD863.PDF.

E. W. Dijkstra. 1995a. Why American computing science seems incurable. EWD1209. https://www.cs.utexas.edu/∼EWD/ewd12xx/EWD1209.PDF.

E. W. Dijkstra. 1995b. A somewhat open letter to David Gries. EWD1227. https://www.cs.utexas.edu/∼EWD/ewd12xx/EWD1227.PDF.

R. M. Dijkstra. 1996. “Everywhere” in predicate algebra and modal logic. *Inform. Process. Lett.* 58, 237–243. DOI: https://doi.org/10.1016/0020-0190(96)00055-5.

E. W. Dijkstra. 2002. EWD 1308: What led to “Notes on structured programming.” In M. Broy and E. Denert (Eds.), *Software Pioneers*. Springer, Berlin, 340–346. DOI: https://doi.org/10.1007/978-3-642-59412-0_19.

E. W. Dijkstra and C. S. Scholten. 1982. About predicate transformers in general. EWD813. https://www.cs.utexas.edu/∼EWD/ewd08xx/EWD813.PDF.

E. W. Dijkstra and C. S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer Verlag, New York. ISBN 9781461232285. DOI: https://doi.org/10.1007/978-1-4612-3228-5.

W. H. J. Feijen and A. J. M. van Gasteren. 1999. *On a Method of Multiprogramming*. Springer Verlag, New York. ISBN 978-1-4419-3179-5. DOI: https://doi.org/10.1007/978-1-4757-3126-2.

W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra (Eds.). 1990. *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*. Springer Verlag. ISBN 0-387-97299-4, 3-540-97299-4. DOI: https://doi.org/10.1007/978-1-4612-4476-9.

R. W. Floyd. 1967. Assigning meanings to programs. In J. Schwartz (Ed.), *Proc. Symp. on Mathematical Aspects of Computer Science*. American Mathematical Society, 19–32. ISBN 0821867288.

G. K. H. Gentzen. 1934. *Untersuchungen Uber Das Logische Schliessen*. Ph.D. thesis, University of Gottingen. An English translation titled “Investigations into Logical Deduction” appears in M. E. Szabo. *The Collected Works of Gerhard Gentzen*. North-Holland Publishing Company, 1969.

D. Gries. 1971. *Compiler Construction for Digital Computers*. John Wiley and Sons, New York.

D. Gries. 1981. *The Science of Programming*. Springer Verlag, New York. DOI: https://doi.org/10.1007/978-1-4612-5983-1.

D. Gries. 1982. A note on a standard strategy for developing loop invariants and loops. *Sci. Comput. Program.* 2, 3, 207–214. DOI: https://doi.org/10.1016/0167-6423(83)90015-1.

D. Gries and F. B. Schneider. 1993. *A Logical Approach to Discrete Math*. Springer Verlag, New York. ISBN 978-0-387-94115-8. DOI: https://doi.org/10.1007/978-1-4757-3837-7.

D. Gries and F. B. Schneider. 1998. Adding the everywhere operator to propositional logic. *J. Log. Comput.* 8, 1, 119–129. DOI: https://doi.org/10.1093/logcom/8.1.119.

D. Gries, M. Paul, and H. R. Wiehle. 1965. Some techniques used in the ALCOR ILLINOIS 7090. *Commun. ACM* 8, 8, 496–500. DOI: https://doi.org/10.1145/365474.365511.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10, 576–580. DOI: https://doi.org/10.1145/363235.363259.

A. Kaldewaij. 1990. *Programming: The Derivation of Algorithms*. Prentice-Hall International, Englewood Cliffs, NJ.

L. Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* SE-3, 2, 125–143. DOI: https://doi.org/10.1109/TSE.1977.229904.

J. Łukasiewicz. 1970. The equivalential calculus. In L. Borkowski (Ed.), *Jan £ukasiewicz: Selected Works*. North-Holland Publishing Company, Amsterdam and London. Appeared first in 1934.

C. Morgan. 1994. *Programming from Specifications* (2nd. ed.). Prentice-Hall International, London.

P. Naur. 1966. Proofs of algorithms by general snapshots. *BIT Numer. Math* 6, 310–316. DOI: https://doi.org/10.1007/BF01966091.

P. Naur and B. Randell (Eds). 1969. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch-Partenkirchen, Germany, 7–11 October 1968*. In J. M. Buxton, P. Naur, and B. Randell (Eds.), *Scientific Affairs Division, NATO, Brussels. Reprinted in Software Engineering: Concepts and Techniques*. Petrocelli/Charter, New York, 1976. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

S. Owicki. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Ph.D. thesis. Cornell University.

S. Owicki and D. Gries. 1976a. An axiomatic proof technique for parallel programs. *Acta Inform*. 6, 319–340. DOI: https://doi.org/10.1007/BF00268134.

S. Owicki and D. Gries. 1976b. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM* 19, 5, 279–285. DOI: https://doi.org/10.1145/360051.360224.

J. Painter and J. McCarthy. 1967. Correctness of a compiler for arithmetic expressions. In J. Schwartz (Ed.), *Proc. Symp. on Mathematical Aspects of Computer Science*. American Mathematical Society, 33–46. ISBN 0821867288.

F. B. Schneider. 1997. *On Concurrent Programming*. Springer Verlag, New York. ISBN 978-1-4419-3179-5. DOI: https://doi.org/10.1007/978-1-4612-1830-2.

J. S. Warford, D. Vega, and S. M. Staley. 2020. A calculational deductive system for linear temporal logic. *ACM Comput. Surv*. 53, 3, 1–38. DOI: https://doi.org/10.1145/3387109.

N. Wirth. 1971. Program development by stepwise refinement. *Commun. ACM* 14, 4, 221–227. DOI: https://doi.org/10.1145/362575.362577.

8 Nondeterminism and Guarded Commands

Krzysztof R. Apt and Ernst-Rüdiger Olderog

8.1 Introduction

The purpose of this chapter is to review Dijkstra's contribution to nondeterminism by discussing the relevance and impact of his guarded commands language proposal. To properly appreciate it we explain first the role of nondeterminism in computer science at the time his original article [Dijkstra 1975] appeared.

The notion of computability is central to computer science. It was studied first in mathematical logic in the 1930s. Several formalisms that aimed at capturing this notion were then proposed and proved equivalent in their expressive power: μ-recursive functions, lambda calculus, and Turing machines, to mention the main ones.

Alan Turing alluded to nondeterminism in his original article on his machines, writing

> For some purposes we might use machines (choice machines or *c*-machines) whose motion is only partially determined by the configuration. [. . .] When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator [Turing 1937, p. 232].

However, subsequently he limited his exposition to deterministic machines, and it seems that the above option has not been pursued for quite some time. As pointed out in Armoni and Ben-Ari [2009], an interesting account of nondeterminism, in the classic book by Martin Davis, Turing machines were used with a restriction that

> no Turing machine will ever be confronted with two different instructions at the same time [Davis 1958, p. 5].

Another early book by Hermes [1965] introduced the theory of computability with deterministic Turing machines as equivalent to the μ-recursive functions. Further, as pointed out in Spaan et al. [1989], another helpful study of nondeterminism, in a standard comprehensive introduction to the recursion theory by Hartley Rogers Jr., Turing machines were explicitly assumed to be deterministic:

> [...] Finally, the device is to be constructed that it behaves according to a finite list of deterministic rules [...] [Rogers Jr. 1987, p. 13].

It seems that a systematic study of addition of nondeterminism to formalisms concerned with computability is due to computer scientists. What follows is a short exposition of such formalisms. Then we discuss Dijkstra's contribution and its relevance. We conclude by providing a brief account of other approaches to nondeterminism that followed.

Literature on nondeterminism in computer science is really extensive. The word "nondeterminism" yields 421 hits in the DBLP database, while "nondeterministic" results in 1141 matches. Our intention was not to provide a survey of the subject but rather to sketch a background against which one can adequately assess Dijkstra's contribution to the subject.

When working on our book [Apt et al. 2009], written jointly with Frank de Boer, we found that guarded commands form a natural "glue" that allowed us to connect the chapters on parallel programs, distributed programs, and fairness into a coherent whole. This explains the regular references to this book in the second half of this chapter.

8.2 Avoiding Nondeterminism

We begin with two early formalisms in which nondeterminism is present but the objective is to use them in such a way that it is not visible in the outcome.

8.2.1 Grammars

Mechanisms that are nondeterministic from the very start are *grammars* in the Chomsky [1959] hierarchy. For example, the set of arithmetic expressions with variables x, y, z, operators $+$ and $*$, and brackets "(" and ")" as terminal symbols can be defined by the following context-free grammar using the start symbol S as its only nonterminal:

$$S ::= x \mid y \mid z \mid S + S \mid S * S \mid (S). \tag{8.1}$$

Here it is natural to postulate that in a derivation step $\vdash$, the nonterminal S can be replaced by any of the above right-hand sides. In particular, the arithmetic

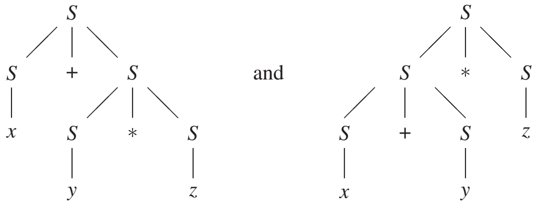


Figure 8.1 Two different parsing trees for $x + y * z$ using grammar (8.1).

expression $x + y * z$ has two different *leftmost derivations*, where always the leftmost occurrence of S is replaced:

$$S \vdash S + S \vdash x + S \vdash x + S * S \vdash x + y * S \vdash x + y * z \tag{8.2}$$

$$S \vdash S * S \vdash S + S * S \vdash x + S * S \vdash x + y * S \vdash x + y * z \tag{8.3}$$

In a compiler, derivation (8.2) corresponds to a parsing tree shown on the left-hand side of Figure 8.1, giving priority to the operator $*$, while derivation (8.3) corresponds to a parsing tree on the right-hand side, giving priority to $+$.

Thus by definition, the above grammar is *ambiguous*. When using context-free grammars to define syntax of a programming language, one is interested in *unambiguous* grammars, meaning that each word (here: program) has only one parsing tree. So in grammars, nondeterminism (in the application of the production rules) is allowed, but the objective is that it does not lead to ambiguities in the above sense.

A context-free grammar that allows nondeterminism but is unambiguous uses three nonterminals, E (for "expression"), T (for "term"), and F (for "factor"), where E is the start symbol, and the following production rules:

$$\begin{aligned} E &::= T \mid E + T \\ T &::= F \mid T * F \\ F &::= (E) \mid x \mid y \mid z. \end{aligned} \tag{8.4}$$

This grammar generates the same set of arithmetic expressions as the one above but is unambiguous. In particular, the arithmetic expression $x + y * z$ has now only one leftmost derivation corresponding to the unique parsing tree shown in Figure 8.2. The grammar encodes the fact that the operator * has a higher priority than + and that expressions with the same operator are evaluated from left to right. It can be generalized to a pattern dealing with any set of infix operators with arbitrary priority among them.

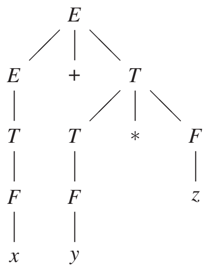


Figure 8.2 Unique parsing tree for $x + y * z$ using grammar (8.4).

8.2.2 Abstract Reduction Systems

Another simple formalism that allows nondeterminism are abstract reduction systems. Formally, an *abstract reduction system* is a pair $(A, \longrightarrow)$ where A is a set and $\longrightarrow$ is a binary relation on A. If $a \longrightarrow b$ holds, we say that a can be *replaced* by b. In this setting nondeterminism means that an element can be replaced in various ways.

There are several important examples of abstract reduction systems, in particular *term rewriting systems*, with *combinatory logic* and λ*-calculus*, and some functional languages as best-known examples (see, e.g., Terese [2003]).

Let $\longrightarrow^*$ denote the reflexive transitive closure of $\longrightarrow$. An element $a \in A$ is said to be in *normal form* if for no $b \in A$, $a \longrightarrow b$ holds. If $a \longrightarrow^* b$ and b is in normal form, then b can be viewed as a *value* of a obtained by means of an abstract computation consisting of a repeated application of the $\longrightarrow$ relation. In general, one is interested in abstract reduction systems in which each element has at most one normal form, so that the notion of a value can be unambiguously defined. One says then that the system has the *unique normal form* (UN, in short).

To establish UN, it suffices to establish the *Church–Rosser property* (CR, in short). It states that for all $a, b, c \in A$

$$\begin{tikzcd}
& a \arrow[dl, "*"'] \arrow[dr, "*"] & \\
b & & c
\end{tikzcd}$$

implies that for some $d \in A$

$$\begin{tikzcd}
b \arrow[dr, "*"'] & & c \arrow[dl, "*"] \\
& d. &
\end{tikzcd}$$

Indeed, CR implies UN.

Several important term rewriting systems, including combinatory logic and λ-calculus, have CR, and hence UN, see, for example, Terese [2003].

In this area the interest in UN means that one is interested in deterministic outcomes *in spite of* the nondeterminism that is present, that is, one aims at showing that the nondeterminism is *inessential*.

8.3 Angelic Nondeterminism

We now proceed with a discussion of formalisms in which addition of nondeterminism allowed one to extend their expressiveness. These formalisms share a characteristic that one identifies successes and failures and only the former count. This kind of nondeterminism was later termed *angelic nondeterminism*.

8.3.1 Nondeterministic Finite Automata and Turing Machines

The first definitions of finite-state automata required deterministic transition functions, as noted in Hopcroft and Ullman [1979]. It took the insight of Rabin and Scott [1959] to introduce *nondeterministic* finite automata in their seminal paper. Such an automaton has choices in its moves: at each transition it may select one of several possible next states. They motivated their definition as follows: "The main advantage of these machines is the small number of internal states that they require in many cases and the ease in which specific machines can be described." They proved by their famous *power set construction* (that uses as the set of states the powerset of the original set of states) that for each nondeterministic automaton a deterministic one can be constructed that accepts the same set of finite words; however, the deterministic one may have exponentially more states. Crucial here is their definition of acceptance: a nondeterministic automaton *accepts* a word if there is *some* successful run of the automaton from an initial to a final state, processing the word symbol by symbol. Thus, only a success counts, while failures do not matter.

This kind of nondeterminism has also been introduced for other types of machines, in particular pushdown automata and Turing machines, leading to the definition of various complexity classes discussed at the end of this section.

8.3.2 McCarthy's Ambiguity Operator

Probably the first proposal to add nondeterminism to a programming language is due to John McCarthy, who introduced in McCarthy [1963] an "ambiguity operator" $amb(x, y)$ that, given two expressions x and y, nondeterministically returns the value of x or of y when both are defined, and otherwise whichever is defined. (McCarthy did not explain what happens when both x and y are undefined, but the most natural assumption is that $amb(x, y)$ is then undefined as well.) In particular, $amb(1, 2)$ yields 1 or 2.

McCarthy was concerned with the development of a functional language, so in his formulation programs were expressions, possibly defined by recursion. As an example of a program that uses the ambiguity operator, he introduced the function $less(n)$ that assigns to each natural number n any nonnegative integer less than n. The function was defined recursively by:

$$less(n) = amb(n-1, less(n-1)).$$

McCarthy did not discuss this function in detail, but note that its definition involves a subtlety because of its use of undefined values. For the smallest value in its domain, namely 1, we get $less(1) = amb(0, less(0)) = 0$, since by definition $less(0)$ is undefined. For the next value, so 2, we get $less(2) = amb(1, less(1)) = amb(1, 0)$, which yields 0 or 1. Next, $less(3) = amb(2, less(2)) = amb(2, amb(1, 0))$, which yields 0 or 1 or 2, and so on. This may be the first example of a program that uses nondeterminism.

McCarthy used his ambiguity operator to extend computable functions by nondeterminism to what he called *computably ambiguous functions*. His work soon inspired first proposals of systems and programming languages that incorporated some form of nondeterminism, mainly to express concisely search problems, see, for example, Smith and Enes [1973] for an account of some of them.

Sometime later, the authors of Zabih et al. [1987] extended a dialect of Lisp with McCarthy's nondeterministic operator denoted by `AMB`. The addition of nondeterminism was coupled with a dependency-directed backtracking, triggered by a special expression `FAIL` that has no value. The resulting language was called SCHEMER. This addition of nondeterminism to Lisp was later discussed in the book by Abelson and Sussman [1996] that used a dialect of Lisp called Scheme.

8.3.3 Floyd's Approach to Nondeterministic Programming

Another approach to programming that corresponds to the type of nondeterminism used in the nondeterministic Turing machines was proposed by Robert Floyd [1967]. He began his article by a fitting quotation from a famous poem "The Road Not Taken" by Robert Frost:

> Two roads diverged in a yellow wood,
> And sorry I could not travel both
> And be one traveler, long I stood
> And looked down one as far as I could
> To where it bent in the undergrowth;

Floyd's idea was to add nondeterminism to the conventional flowcharts by

— using a nondeterministic assignment

$$x := choice(t),$$

that assigns to the variable x an arbitrary positive integer of at most the value of the integer expression t, and

— labeling all termination points as *success* or *failure*.

Thus, augmented flowcharts can generate several execution sequences: however, only those that terminate in a node labeled *success* are considered as the computations of the presented algorithm.

Using a couple of examples, Floyd showed how these two additions can lead to simple flowchart programs that search for a solution through exhaustive enumeration and which otherwise would have to be programmed using backtracking combined with appropriate bookkeeping.

Jacques Cohen [1979] illustrated Floyd's approach using conventional programs in which one uses the above nondeterministic assignment statement and a **fail** statement that corresponds to a node labeled *failure*. Any computation that reaches the **fail** statement terminates improperly (it *aborts*). The label *success* is unneeded as it is implicitly modeled by a terminating computation that does not abort. We call such computations *successful*. In this approach, a program is correct if *some* successful computation establishes the assumed postcondition.

We illustrate this approach by means of one of Floyd's examples, the problem of *eight queens* in which one is asked to place eight queens on the chessboard so that they do not attack each other. The program looks as follows, where a, b, and c are integer arrays with appropriate bounds and initialized for all indices to 0:

```
for $col := 1$ to 8 do
  $row := choice(8)$;
  if $a[row] = 1 \vee b[row + col] = 1 \vee c[row - col] = 1$ then fail fi;
  $a[row] := 1$;
  $b[row + col] := 1$;
  $c[row - col] := 1$
od
```

Subscripted variables have the following interpretation:

$a[i] = 1$ means that a queen was placed in the ith row,

$b[i] = 1$ means that a queen was placed in the ith ↘ diagonal,

$c[i] = 1$ means that a queen was placed in the ith ↙ diagonal,

where the ↘ diagonals are the ones for which the sum of the coordinates is the same and the ↙ diagonals are the ones for which the difference of the coordinates is the same. Upon successful termination a solution is produced in the form of a sequence of eight values that are successively assigned to the variable *row*; these values correspond to the placements of the queens in the columns 1 to 8.

Thanks to the nondeterministic assignment and the **fail** statement, this program can generate several computations, including ones that abort. The successful computations generate precisely all solutions to the eight queens problem.

Floyd showed that one can convert his augmented flowcharts to conventional flowcharts by means of a generic transformation that boils down to an implementation of backtracking. This way one obtains a deterministic algorithm that generates a solution to the considered problem, and it is easy to modify his transformation so that all solutions are generated. Of course, such a transformation can also be defined for the programs considered here.

8.3.4 Logic Programming and Prolog

In the early seventies angelic nondeterminism was embraced in a novel approach to programming that combined the idea of *automatic theorem proving* based on the use of relations, with built-in automatic backtracking. It was realized in the programming language Prolog, conceived and implemented by Alain Colmerauer and his team (see the historic account in Colmerauer and Roussel [1996]), while its theoretical underpinnings, called *logic programming*, were provided by Robert Kowalski [1974].

A detailed discussion of logic programming and Prolog is out of scope of this chapter, but to illustrate the nondeterminism present in this approach to programming consider a simple logic programming program that appends two lists. It is defined by means of two *clauses*, the first one unconditional and the second one conditional:

```
append([], Xs, Xs).
append([X | Xs], Ys, [X | Zs]) ← append(Xs, Ys, Zs).
```

Here `append` is a name of a relation, `X`, `Xs`, `Ys`, and `Zs` are the variables, `[]` denotes the empty list, and `[X | Xs]` denotes the list with the head `X` and the tail `Xs`.

The first clause states that the result of appending `[]` and the list `Xs` is `Xs`. The second clause is a reverse implication stating that if the result of appending the lists `Xs` and `Ys` is `Zs`, then the result of appending the lists `[X | Xs]` and `Ys` is `[X | Zs]`.

In general, a program is a set of clauses that are built from *atomic queries*. In the above program, atomic queries are of the form `append(s, t, u)`, where `s, t, u` are expressions built out of the variables and the constant `[]` using the list formation operation `[.|.]`. A *query* is a conjunction of atomic queries. A program is activated by executing a query, which is a request to evaluate it w.r.t. the considered program. We do not discuss here the underlying computation model; it suffices to know that the computation searches for an instance of the query that logically follows from the program. If such an instance is found, one says that a query *succeeds* and otherwise that it *fails*.

In logic programming nondeterminism arises in two ways, by the fact that relations can be defined using several clauses and by the choice of the atomic query to be evaluated first. At the abstract level this is the same form of nondeterminism as the one used in Floyd's approach: a computation succeeds if at all choice points the right selections are made.

Since a query can succeed in several ways, several solutions can be generated. Consider, for example, the query

```
append(_, Zs, [mon, tue, wed, thu, fri]), append(Xs, _, Zs),
```

in which we use Prolog's convention according to which each occurrence of '_' stands for a different (*anonymous*, i.e., irrelevant) variable and the comma is used (between the atomic queries) instead of the conjunction sign. Intuitively, this query stipulates that `Xs` is a prefix of a suffix of the list `[mon, tue, wed, thu, fri]`. Successful computations of this query w.r.t. the above program generate in `Xs` all possible sublists of this list.

In Prolog, nondeterminism present in the computation model of logic programming is resolved by stipulating that the first clause and the first atomic query from the left are selected, and by providing a built-in automatic backtracking that allows the computation to recover from a failure.

8.3.5 Does Angelic Nondeterminism Add Expressive Power?

After this discussion of nondeterministic programming let us look at the idea of angelic nondeterminism through the lens of computing and structural complexity.

We mentioned already in the introduction that nondeterminism was not considered in the original formalizations of the notion of a computation. But once one considers restricted models of computability, nondeterminism naturally arises. An early example is the characterization of formal languages in the Chomsky hierarchy of formal grammars. It distinguishes four types of grammars that correspond to four levels of formal languages of increasing complexity: regular, context-free, context-sensitive, and recursively enumerable languages, denoted by Type-3, Type-2, Type-1, and Type-0, respectively.

The need for nondeterminism arises in connection with the characterization of these classes by means of an automaton. Whereas Type-0 languages at the top of this hierarchy are the ones accepted by the deterministic Turing machines, and

Type-3 languages at the bottom, that is, regular languages, are the ones accepted by the deterministic finite automata, the characterization of the remaining two levels calls for the use of nondeterministic automata or machines.

In particular Type-2 languages, that is, context-free languages, cannot be accepted by deterministic pushdown automata. A pushdown automaton extends a finite automaton by an unbounded stack or pushdown list that is manipulated in a "first in–last out" fashion while scanning a given input word letter by letter. Such an automaton is called *deterministic* if for each control state each symbol at the top of the stack, and each input symbol, has at most one possible move; it is called *nondeterministic* if more than one move is allowed.

A standard example is the language of all *palindromes* over letters *a* and *b*, that is, words that read the same forward and backward, like *abba*. This language is context-free, that is, it can be generated by a context-free grammar, but it cannot be accepted by a deterministic pushdown automaton. The intuitive reason is that while checking an input word letter by letter, one has to "guess" when the middle of the word has been reached so that one can test that from now on the letters occur in the reverse order, by referring to the constructed pushdown list.

However, context-free languages can be characterized by means of nondeterministic pushdown automata. Such a characterization refers to angelic nondeterminism because it states that a word is generated by the language iff it can be accepted by *some* computation of the automaton.

Similarly, nondeterminism is used to characterize Type-1 languages, that is, context-sensitive languages: they are the ones that can be recognized by linear bounded nondeterministic Turing machines.

While deterministic and nondeterministic Turing machines accept the same class of languages, the Type-0 languages, there is a difference when time complexity is considered. Probably the most known are the complexity classes P and NP of problems that can be solved in polynomial time by deterministic Turing machines and nondeterministic Turing machines, respectively. The class NP was introduced by Stephen Cook [1971] and Leonid Levin [1973]. The intuition is that a problem is in NP if it can be solved by first (nondeterministically) guessing a candidate solution and then checking in polynomial time whether it is indeed a solution. Following the paradigm of angelic nondeterminism, wrong guesses do not count. The famous open problem posed by Cook is whether P = NP holds.

In contrast, when instead of time space is considered as the complexity measure, it is known that there is no difference between the resulting classes PSPACE and NSPACE of problems that can be solved in polynomial space by deterministic Turing machines and nondeterministic Turing machines, respectively. This is a consequence of a result by Savitch [1970].

We conclude that the addition of angelic nondeterminism can, but does not have to, increase expressive power of the considered model of computability.

8.4 Guarded Commands

One of us (KRA) met Edsger Dijkstra for the first time in Spring 1975 while looking for an academic job in computer science in the Netherlands. During a meeting at his office at the Technical University of Eindhoven, Dijkstra handed him a copy of his EWD472 titled "Guarded commands, nondeterminacy and formal derivation of programs." It appeared later that year as Dijkstra [1975]. This short, five-pages long paper introduced two main ideas: a small programming language, now called *guarded commands*, and its semantics, now called the *weakest precondition semantics*. Both were new ideas of great significance.

In this chapter we focus on the guarded commands; Chapter 7, written by David Gries, and Chapter 6, written by Reiner Hähnle, discuss the weakest precondition semantics. The essence of guarded commands boils down to two new programming constructs:

- *alternative command*

$$S ::= \textbf{if } B_1 \longrightarrow S_1 [] \dots [] B_n \longrightarrow S_n \textbf{ fi},$$

- *repetitive command*

$$S ::= \textbf{do } B_1 \longrightarrow S_1 [] \dots [] B_n \longrightarrow S_n \textbf{ od}.$$

We sometimes abbreviate these commands to

$$\textbf{if } []_{i=1}^{n} B_i \longrightarrow S_i \textbf{ fi} \text{ and } \textbf{do } []_{i=1}^{n} B_i \longrightarrow S_i \textbf{ od}.$$

A Boolean expression B_i within S is called a *guard* and the construct $B_i \longrightarrow S_i$ is called a *guarded command*.

The symbol $[]$ represents a nondeterministic choice between the guarded commands $B_i \longrightarrow S_i$. The alternative command

$$\textbf{if } B_1 \longrightarrow S_1 [] \dots [] B_n \longrightarrow S_n \textbf{ fi}$$

is executed by executing a statement S_i for which the corresponding guard B_i evaluates to true. There is no rule saying which statement among those whose guard evaluates to true should be selected. If all guards B_i evaluate to false, the alternative command *aborts*.

The selection of guarded commands in the context of a repetitive command

$$\mathbf{do}\, B_1 \longrightarrow S_1 [] \ldots [] B_n \longrightarrow S_n\, \mathbf{od}$$

is performed in a similar way. The difference is that after termination of a selected statement S_i the whole command is repeated starting with a new evaluation of the guards B_i. Moreover, in contrast to the alternative command, the repetitive command properly terminates when all guards evaluate to false.

Dijkstra did not establish the notation of guarded commands directly. Two earlier EWDs reveal that he first considered other options, also about their intended semantics.

In EWD398 [Dijkstra 1973a] he first used ‘,’ to separate guarded commands, but changed it halfway to [], reporting criticism of Don Knuth and stating “this whole report is an experiment in notation!” Also, he wrote $B : S$ instead of $B \longrightarrow S$ adopted in Dijkstra [1975] and used by him thereafter. Further, for the repetitive command he wrote that

> In the case of more than one executable command, it is again undefined which one will be selected, we postulate, however, that then they will be selected in “fair random order”, i.e. we disallow the non-determinacy permanent neglect of a permanently executable guarded command from the list.

However, a day later, in EWD399 [Dijkstra 1973b], he admitted that this decision

> [. . .] was a mistake: for such constructs we prefer now not to exclude nontermination. It is just too tricky if the termination—and in particular: the proof of the termination—has to rely on the fair randomness of the selection and we had better restrict ourselves to constructs w[h]ere each guarded command, when executed, implies a further approaching of the terminal state.

We shall return to this problem of fairness shortly. But first let us focus on the main feature of guarded commands, the nondeterminism they introduce. However, this nondeterminism is of a different type than the one we discussed so far: by definition a guarded command program establishes the desired postcondition if *all* possible executions establish it. This kind of nondeterminism was later termed *demonic nondeterminism*.

This seems at first sight like a flawed decision: why should one complicate matters by adding to the program more possible executions paths when one will suffice? But, as we shall soon see, there are good reasons for doing it.

An often-cited example in favor of the guarded commands language is the formalization of Euclid's algorithm that computes the greatest common divisor (*gcd*) of two positive integers x and y

$$\mathbf{do}\, x > y \rightarrow x := x - y \;[]\; x < y \rightarrow y := y - x \,\mathbf{od}$$

that terminates with the *gcd* of the initial values of x and y equal to their final, common value.

However, this program is not nondeterministic: for any initial value of the variables x and y there is only one possible program execution. This program actually illustrates something else: that guarded commands allow one to write more elegant algorithms. Here the variables x and y are treated symmetrically, which is not the case when a deterministic program is used.

In another simple example from Dijkstra's paper, one is asked to compute the maximum *max* of two numbers, x and y:

$$\mathbf{if}\, x \geq y \rightarrow max := x \;[]\; y \geq x \rightarrow max := y \,\mathbf{if}$$

It illustrates the nondeterminism in the mildest possible form: when $x = y$ two executions are possible, but the outcome is still the same.

A slightly more involved example of a nondeterministic program with a deterministic outcome is Dijkstra's solution to the following problem: assign to the variables x_1, x_2, x_3, and x_4 an ordered permutation of the values X_1, X_2, X_3, and X_4, that is, one such that $x_1 \leq x_2 \leq x_3 \leq x_4$ holds. The program uses a parallel assignment that forms part of the guarded commands language:

```
$x_1, x_2, x_3, x_4 := X_1, X_2, X_3, X_4$;
do $x_1 > x_2 \rightarrow x_1, x_2 := x_2, x_1$
[]  $x_2 > x_3 \rightarrow x_2, x_3 := x_3, x_2$
[]  $x_3 > x_4 \rightarrow x_3, x_4 := x_4, x_3$
od
```

Upon exit all guards evaluate to false, that is, $x_1 \leq x_2 \leq x_3 \leq x_4$ holds, as desired. The relevant invariant is that x_1, x_2, x_3, x_4 is a permutation of X_1, X_2, X_3, X_4,

Finally, the following example of Dijkstra results in a program with a nondeterministic outcome. The problem is to find for a fixed $n > 0$ and a fixed integer-valued function f defined on $\{0, \ldots, n-1\}$ a maximum point of f, that is, a value k such that

$$k \in \{0, \ldots, n-1\} \wedge \forall i \in \{0, \ldots, n-1\} : f(k) \geq f(i).$$

A simple solution is the following program:

```
$k := 0; j := 1;$
do $j \neq n \longrightarrow$ if $f(j) \leq f(k) \longrightarrow j := j + 1$
                 $[\,]\; f(j) \geq f(k) \longrightarrow k := j;\; j := j + 1$
                 fi
od
```

It scans the values of f starting with the argument 0, updates the value of k in case a new maximum is found (when $f(j) < f(k)$), and optionally updates the value of k in case another current maximum is found (when $f(j) = f(k)$). The relevant invariant is here

$$k \in \{0, \ldots, j-1\} \wedge j \leq n \wedge \forall i \in \{0, \ldots, j-1\} : f(k) \geq f(i).$$

Indeed, it is established by the initial assignment, maintained by each loop iteration, and upon termination it implies the desired condition on k. Note that the program can compute in k any maximum point of f.

All these small examples (and there are no others in Dijkstra's paper) do not provide convincing reasons for embracing nondeterminism provided by the guarded commands language. A year after Dijkstra [1975] appeared, Dijkstra published his book [Dijkstra 1976] in which he derived several elegant guarded command programs, including a more efficient version of the above program that avoids the recomputation of $f(j)$ and $f(k)$. But inspecting these programs we found only a few examples in which guards were not mutually exclusive and only two programs with a nondeterministic outcome.

So why then has demonic nondeterminism, as present in the guarded command language, turned out to be so influential? In what follows we discuss subsequent developments that provide some answers to this question. Many accounts of the guarded command language discuss it, as Dijkstra originally did, together with its weakest precondition semantics. But to appreciate the nondeterminism Dijkstra introduced, in our view it is useful to separate the language from its weakest precondition semantics.

8.5 Some Considerations on Guarded Commands

Dijkstra's famous article "Go to statement considered harmful" [Dijkstra 1968a] shows that he was aware of an ancestor of his alternative command in the form of a conditional expression $(B_1 \longrightarrow e_1, \ldots, B_n \longrightarrow e_n)$ introduced by McCarthy [1963]. Here B_is are Boolean expressions and e_is are expressions. The Boolean expressions do not need to be mutually exclusive, but the conditional expressions are deterministic: when executed, the $(B_1 \longrightarrow e_1, \ldots B_n \longrightarrow e_n)$ yields the value of the first expression e_i for which B_i evaluates to true. When all B_is evaluate to false, the conditional expression is supposed to be undefined. So $(B_1 \longrightarrow e_1, \ldots B_n \longrightarrow e_n)$ is a shorthand for a nested **if-then-else** statement.

Dijkstra's explicit introduction of an abort, as opposed to McCarthy's reference to "undefined," is useful because it provides a simple way of implementing an **assert** *B* statement that checks whether assertion *B* holds and causes an abort when this is not the case.

McCarthy worked within the framework of a functional language, so he was constrained to use recursion instead of a looping construct. As a result, Euclid's algorithm is formalized in his notation as follows:

$$gcd(m,n) = (m > n \longrightarrow gcd(m-n,n),\ n > m \longrightarrow gcd(m,n-m),\ m = n \longrightarrow m),$$

which is less elegant than Dijkstra's solution due to the need for the final component of the conditional expression.

In contrast, as already explained, Dijkstra did not prescribe any order in which the guards are selected and ensured that his repetitive command was not defined using the alternative command. In Dijkstra [1975], he motivated his introduction of nondeterminism (called by him nondeterminacy) as follows:

> Having worked mainly with hardly self-checking hardware, with which nonreproducing behavior of user programs is a very strong indication of a machine malfunctioning, I had to overcome a considerable mental resistance before I found myself willing to consider nondeterministic programs seriously. [...] Whether nondeterminacy is eventually removed mechanically—in order not to mislead the maintenance engineer—or (perhaps only partly) by the programmer himself because, at second thought, he does care—e.g., for reasons of efficiency—which alternative is chosen is something I leave entirely to the circumstances. In any case we can appreciate the nondeterministic program as a helpful stepping stone.

But soon he overcame this resistance and one year later he wrote:

> Eventually, I came to regard nondeterminacy as the normal situation, determinacy being reduced to a—not even very interesting—special case. [Dijkstra 1976, p. xv]

McCarthy's semantics of conditional expression can be viewed as an example of such a "mechanical removal" of nondeterminism. However, keeping nondeterminism intact often leads to simpler and more natural programs even if the outcome is deterministic. In some programs the considered alternatives do not need to be mutually exclusive as long as all cases are covered.

A beautiful example was provided by David Gries [1981] in his book. Consider the following problem due to Wim Feijen. (We follow here the presentation of Gries.)

Given are three magnetic tapes, each containing a list of different names in alphabetical order. The first contains the names of people working at IBM Yorktown Heights, the second the names of students at Columbia University, and the third the names of people on welfare in New York City. It is known that at least one person is on all three lists. The problem is to locate the alphabetically first such person.

In Gries [1981], the following elegant program solving this problem was systematically derived. We assume here that the lists of names are given in the form of ordered arrays $a[0:M]$, $b[0:M]$, and $c[0:M]$:

```
$i := 0; j := 0;\ k := 0;$
do $a[i] < b[j] \rightarrow i := i + 1$
[] $b[j] < c[k] \rightarrow j := j + 1$
[] $c[k] < a[i] \rightarrow k := k + 1$
od
```

Note that upon termination of the loop $a[i] = b[j] = c[k]$ holds. The appropriate invariant is

$$0 \leq i \leq i_0 \ \wedge\ 0 \leq j \leq j_0 \ \wedge\ 0 \leq k \leq k_0 \ \wedge\ r$$

where r states that the arrays $a[0:M]$, $b[0:M]$, and $c[0:M]$ are ordered, with $i_0, j_0, k_0 \leq M$, and (i_0, j_0, k_0) is the lexicographically smallest triple such that $a[i_0] = b[j_0] = c[k_0]$.

This program uses nondeterministic guards, so various computations are possible. Still, it has a deterministic outcome.

In general, as soon as two or more guards are used in a loop, in the customary, deterministic, version of the program one is forced to use a, possibly nested, **if-then-else** statement, like in McCarthy's "determinisation" approach. It imposes an evaluation order of the guards, destroys symmetry between them, and does not make the resulting programs easier to verify.

8.6 Modeling Parallel Programs

Concurrent programs, introduced in Dijkstra's [1968b] "Cooperating sequential processes" paper, can share variables, which makes it difficult to reason about them. Therefore, starting with Ashcroft and Manna [1971] and Flon and Suzuki [1978, 1981], various authors proposed to analyze them at the level of nondeterministic programs, where the nondeterminism reflects the existence of various component programs. Such a reduction is possible if one assumes that no concurrent reading and writing of variables takes place.

Using guarded commands, it is possible to make the link between these two classes of programs explicit by a transformation. The precise transformation is a bit laborious, see Flon and Suzuki [1978], so we illustrate it by an example taken from Apt et al. [2009]. Consider the following concurrent program due to Owicki and Gries [1976] that searches for a positive value in an integer array $ia[0:N]$:

```
$i := 1;\ j := 2;\ oddtop := N+1;\ eventop := N+1;$
$[S_1 \| S_2];$
$k := min(oddtop, eventop),$
```

where S_1 and S_2 are deterministic components S_1 and S_2 scanning the odd and the even subscripts of *ia*, respectively:

```
$S_1 \equiv a$: while $i < min(oddtop, eventop)$ do
              $b$: if $ia[i] > 0$ then $c$: $oddtop := i$ else $d$: $i := i + 2$ fi
        od
```

and

```
$S_2 \equiv a$: while $j < min(oddtop, eventop)$ do
              $b$: if $ia[j] > 0$ then $c$: $eventop := j$ else $d$: $j := j + 2$ fi
        od
```

Upon termination of both components, the minimum of two shared integer variables *oddtop* and *eventop* is checked. The labels a, b, c, d, and e are added here to clarify the transformation. The parallel composition $S \equiv [S_1 \| S_2]$ is transformed into the following guarded commands program $T(S)$ with a single repetitive command that employs the control variables cv_1 and cv_2 for S_1 and S_2 that can assume the values of the labels:

```
$T(S) \equiv cv_1 := a;\ cv_2 := a;$
        do $cv_1 = a \wedge i < min(oddtop, eventop) \longrightarrow cv_1 := b$
        [] $cv_1 = a \wedge \neg(i < min(oddtop, eventop)) \longrightarrow cv_1 := e$
        [] $cv_1 = b \wedge ia[i] > 0 \longrightarrow cv_1 := c$
        [] $cv_1 = b \wedge \neg(ia[i] > 0) \longrightarrow cv_1 := d$
        [] $cv_1 = c \longrightarrow oddtop := i;\ cv_1 := a$
        [] $cv_1 = d \longrightarrow i := i + 2;\ cv_1 := a$
        [] $cv_2 = a \wedge j < min(oddtop, eventop) \longrightarrow cv_2 := b$
        [] $cv_2 = a \wedge \neg(j < min(oddtop, eventop)) \longrightarrow cv_2 := e$
        [] $cv_2 = b \wedge ia[j] > 0 \longrightarrow cv_2 := c$
[] $cv_2 = b \wedge \neg(ia[j] > 0) \longrightarrow cv_2 := d$
[] $cv_2 = c \longrightarrow eventop := j;\ cv_2 := a$
[] $cv_2 = d \longrightarrow j := j + 2;\ cv_2 := a$
od;
if $cv_1 = e \wedge cv_2 = e \longrightarrow skip$ fi
```

Note that the repetitive command exhibits nondeterminism. For example, when $cv_1 = cv_2 = a$, two guarded commands can be chosen next. This corresponds to the *interleaving semantics* of concurrency that we assume here. When the repetitive command has terminated, the final alternative command checks whether this termination is the one intended by the original concurrent program S. This is the case when both cv_1 and cv_2 store the value e. In the current example, this check is trivially satisfied and thus the alternative command could be omitted.

However, for concurrent programs with synchronization primitives, a termination of the repetitive command may be due to a deadlock in the concurrent program. Then the final alternative command is used to transform the deadlock into a failure, indicating an undesirable state at the level of nondeterministic programs. For details of this transformation, we refer to chapter 10 of Apt et al. [2009].

This transformation allows us to clarify that the nondeterminism resulting from parallelism is the one used in the guarded commands language. However, this example also reveals a drawback of the transformation: the structure of the original parallel program gets lost. The resulting nondeterministic program represents a single loop at the level of an assembly language with atomic actions explicitly listed. The assignments to the control variables correspond to **go to** statements, which explains why reasoning about the resulting program is difficult. Interestingly, this problem does not arise for the transformation of the Communicating Sequential Processes programs that we give in the next section.

8.7 Communicating Sequential Processes and Their Relation to Guarded Commands

Dijkstra's quoted statement, "In any case we can appreciate the nondeterministic program as a helpful stepping stone," suggests that he envisaged some extensions of the guarded command language. But in his book [Dijkstra 1976] he only augmented it with local variables by providing an extensive notation for various uses of local and global variables, and added arrays. In his subsequent research he only used the resulting language.

However, his discussion of the guarded commands program formalizing Euclid's algorithm suggests that he also envisaged some connection with concurrency. He suggested that the program could be viewed as a synchronization of two cyclic processes **do** $x := x - y$ **od** and **do** $y := y - x$ **od** in such a way that the relation $x > 0 \wedge y > 0$ is kept invariantly true. Still, he did not pursue this idea further.

Subsequent research showed that guarded command programs can be viewed as a natural layer lying between deterministic and concurrent programs. This was first made clear in 1978 by Tony Hoare who introduced in Hoare [1978] an elegant language proposal for distributed programming that he called Communicating Sequential Processes (abbreviated to CSP) in clear reference to Dijkstra's "Cooperating sequential processes" paper [Dijkstra 1968b].

Hoare stated seven essential aspects of his proposal, mentioning as the first one Dijkstra's guarded commands "as the sole means of introducing and controlling nondeterminism." The second one also referred to Dijkstra, namely to his parallel command, according to which, "All the processes start simultaneously, and the parallel command ends only when they are all finished." It is useful to discuss CSP in some detail to see how each of these two aspects results in the same type of nondeterminism.

In Dijkstra's cooperating sequential processes model, processes communicate with each other by updating global variables. By contrast, in CSP processes communicate solely by means of the *input* and *output* commands, which are atomic statements that are executed in a synchronized fashion. So CSP processes do not share variables.

For the purpose of communication, CSP processes have names. The input command has the form $P?x$, which is a request to process (named) P to provide a value to the variable x, while the output command has the form $Q!t$, which is a granting of the value of the expression t to process (named) Q. When the types of x and t match and the processes refer to each other, we say that the considered input and output commands *correspond*. They are then executed simultaneously; the effect is that of executing the assignment $x := t$.[1]

In CSP, a single input command is also allowed to be part of a guard. The restriction to input commands was dictated by implementation considerations. But once it was clarified how to implement the use of output commands in guards, they were admitted as part of a guard as well. So, in the sequel we admit both input and output commands in guards. Thus, guards are of the form $B; \alpha$, where B is a Boolean expression and α is an input or output command, *i/o command* for short.

[1] Note that not all assignments can be modeled this way. For instance, the assignment $x := x + 1$ cannot be reproduced since processes do not share variables.

We assume that such an extended guard *fails* when the Boolean part evaluates to false.[2]

To illustrate the language, consider an example, taken from Apt et al. [2009], which is a modified version of an example given in Hoare [1978]. In what follows we refer to the repetitive commands of CSP as **do** loops.

We wish to transmit a sequence of characters from the process *SENDER* to the process *RECEIVER* with all blank characters (represented by ‘ ’) deleted. To this end we employ an intermediary process *FILTER* and consider a distributed program

$$[SENDER \,\|\, FILTER \,\|\, RECEIVER].$$

The sequence of characters is initially stored by the process *SENDER* in the array $a[0 : M-1]$ of characters, with ‘$*$’ as the last character. The process *FILTER* uses an array $b[0 : M-1]$ of characters as an intermediate store for processing the character sequence and the process *RECEIVER* has an array $c[0 : M-1]$ of the same type to store the result of the filtering process. For coordinating its activities, the process *FILTER* uses two integer variables *in* and *out* pointing to elements in the array $a[0 : M-1]$. The processes are defined as follows:

```
$SENDER$ ::   $i := 0$; do $i \neq M; FILTER\,!\,a[i] \longrightarrow i := i+1$ od

$FILTER$ ::   $in := 0;\ out := 0;\ x := $ ‘ ’;
              do $x \neq$ ‘$*$’$; SENDER\,?\,x \longrightarrow$
                   if $x =$ ‘ ’ $\longrightarrow skip$
                   [] $x \neq$ ‘ ’ $\longrightarrow b[in] := x;\ in := in+1$
                   fi
              [] $out \neq in; RECEIVER\,!\,b[out] \longrightarrow out := out+1$
              od

$RECEIVER$ :: $j := 0;\ y := $ ‘ ’;
              do $y \neq$ ‘$*$’$; FILTER\,?\,y \longrightarrow c[j] := y;\ j := j+1$ od
```

Note that the processes *SENDER* and *RECEIVER* are deterministic, in the sense that each extended guarded command either has just one guard or the Boolean parts of the used guards are mutually exclusive (this second case does not occur here), while *FILTER* is nondeterministic as it uses a **do** loop with two extended guards, the Boolean parts of which are not mutually exclusive. They represent two possible actions for *FILTER* : to communicate with *SENDER* or with *RECEIVER*.

Hoare presented in his article several elegant examples of CSP programs. In some of them the processes are deterministic. But even then, if there are four or

[2] Hoare also allowed an extended guard to fail when its i/o command refers to a process that terminated. For simplicity, do not adopt this assumption here.

more processes, the resulting program is nondeterministic since it admits more than one computation.

By assumption a CSP program is correct if all of its computations properly terminate in a state that satisfies the assumed postcondition. So, this is exactly demonic nondeterminism, like in the case of parallel programs.

This makes it possible to translate CSP programs in a simple way to guarded command programs. In Apt et al. [2009] we provided such a transformation for a fragment of CSP, in which the above example is written. The CSP programs in this fragment are of the form

$$S \equiv [S_1 \parallel \ldots \parallel S_n],$$

where each process S_i is of the form

$$S_i \equiv S_{i,0};\ \mathbf{do}\ []_{j=1}^{m_i} B_{i,j};\alpha_{i,j} \longrightarrow S_{i,j}\ \mathbf{od},$$

each $S_{i,j}$ is a guarded command program, each $B_{i,j}$ is a Boolean expression, and each $\alpha_{i,j}$ is an i/o command. So each process S_i has a single **do** loop in which i/o commands appear only in the guards. No further i/o commands are allowed in the initialization part $S_{i,0}$ or in the bodies $S_{i,j}$ of the guarded commands.

As shown in Apt et al. [1987] and Zöbel [1988], each CSP program can be transformed into a program in this fragment by introducing some control variables.

As abbreviation we introduce

$$\Gamma = \{(i,j,r,s) \mid \alpha_{i,j} \text{ and } \alpha_{r,s} \text{ correspond and } i < r\}.$$

According to the CSP semantics, two generalized guards from different processes can be passed jointly when their i/o commands correspond and their Boolean parts evaluate to true. Then the communication between the i/o commands takes place. The effect of a communication between two corresponding i/o commands $\alpha_1 \equiv P?x$ and $\alpha_2 \equiv Q!t$ is the assignment $x := t$. Formally, for two such commands we define

$$\mathit{Eff}(\alpha_1, \alpha_2) \equiv \mathit{Eff}(\alpha_2, \alpha_1) \equiv x := t.$$

We transform S into the following guarded commands program $T(S)$:

$$\begin{aligned} T(S) \ \equiv\ & S_{1,0};\ \ldots;\ S_{n,0}; \\ & \mathbf{do}\ \ []_{(i,j,r,s)\in\Gamma}\, B_{i,j} \wedge B_{r,s} \longrightarrow \mathit{Eff}(\alpha_{i,j}, \alpha_{r,s}); \\ & \qquad\qquad\qquad\qquad\qquad\qquad S_{i,j};\ S_{r,s} \\ & \mathbf{od}, \end{aligned}$$

where we use of elements of Γ to list all guards in the **do** loop. In the degenerate case when $\Gamma = \emptyset$, we drop this loop from $T(S)$.

For example, for $SFR \equiv [SENDER\|FILTER\|RECEIVER]$ we obtain the following guarded commands program:

```
T(SFR) ≡ $i := 0;\ in := 0;\ out := 0;\ x := `\ ';\ j := 0;\ y := `\ ';$
         do $i \neq M \wedge x \neq `*' \longrightarrow x := a[i];\ i := i + 1;$
                                   if $x = `\ ' \longrightarrow skip$
                                   [] $x \neq `\ ' \longrightarrow b[in] := x;\ in := in + 1$
                                   fi
         [] $out \neq in \wedge y \neq `*' \longrightarrow y := b[out];\ out := out + 1;\ c[j] := y;\ j := j + 1$
         od
```

The semantics of S and $T(S)$ are not identical because their termination behavior is different. However, the final states of properly terminating computations of S and the final states of properly terminating computations of $T(S)$ that satisfy the condition

$$TERM \equiv \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m_i} \neg B_{i,j}$$

coincide. An interested reader can consult chapter 11 of [Apt et al. 2009].

The above transformation makes precise what we already mentioned when discussing an example CSP program: the CSP language introduces nondeterminism in two ways. The first one comes from allowing guarded commands; in the transformed program these are the programs $S_{a,b}$. The second one results from synchronous communication, modeled in the transformation by means of the outer repetitive command. So both ways are instances of demonic nondeterminism.

Thanks to the special form of CSP programs, the transformed program does not introduce any new variables. As a result, this transformation suggests a simple way to reason about the correctness of CSP programs by considering the translated guarded commands program, see chapter 11 of [Apt et al. 2009].

8.8 Fairness

One of the programs in Dijkstra [1976] with a nondeterministic outcome is the following one:

```
$goon :=$ true; $x := 1;$
do $goon \longrightarrow x := x + 1$
[] $goon \longrightarrow goon :=$ false
od
```

The problem Dijkstra was addressing was that of writing a program that sets x to any natural number. He concluded using his weakest precondition semantics that no guarded commands program exists that sets x to any natural number *and* always terminates. Note that the above program can set x to an arbitrary natural number but also can diverge.

As noted in Plotkin [1976], Dijkstra's conclusion can be obtained in a more direct way by appealing to any operational semantics that formalizes the notion of a computation—by representing the computations in the form of a tree. Branching models the execution of a guarded command; each branch corresponds to a successful evaluation of a guard. Given an input state of a guarded command program we have then a finitely branching tree representing all possible computations. Let us call it a *computation tree*.

Denote a program that sets x to any natural number by x :=?. Suppose that it can be represented by a guarded commands program S. Then for any input state the computations of S form a finitely branching computation tree with infinitely many leaves, each of them corresponding to a different natural number assigned to x. We can now appeal to König's [1927] lemma which states that any finitely branching infinite tree has an infinite path. It implies that in every computation tree of S an infinite computation exists, that is, that for every input state the program S can diverge.

Dijkstra's proof contained in chapter 9 of Dijkstra [1976] proceeds differently. He first showed that his predicate transformer *wp* is continuous in the predicate argument and then that the specification of x :=? in terms of *wp* contradicts continuity. In his terminology, the program x :=? introduces *unbounded nondeterminism*, which means that no *a priori* upper bound for the final value of x can be given.

The program x :=? occupied his attention a number of times. In EWD673 published as Dijkstra [1982b], he noted that to prove termination of guarded command programs augmented with the program x :=? it does not suffice to use integer-valued bound functions and one has to resort to well-founded relations. In turn, in EWD675 published as Dijkstra [1982c], he noticed that the converse implication holds as well: the existence of a program for which *wp* is not continuous implies the existence of a program with unbounded nondeterminism.

Soon, more in-depth studies of the program x :=? and the consequences of its addition to deterministic programs followed. In particular, Chandra [1978] showed that the halting problem for programs admitting x :=? is Π^1_1 complete, so of higher complexity than the customary halting problem for computable functions (see also Spaan et al. [1989] for a further discussion of this problem), while Back [1980] and Boom [1982] advocated use of x :=? as a convenient form of program abstraction that deliberately ignores the details of an implementation. One may note here that Hilbert's ε notation essentially serves a similar purpose: $\varepsilon x \phi$ picks an x that satisfies the formula ϕ. We wish to remain within the realm of guarded commands, so we limit ourselves to a clarification of the relation of the program x :=? with the notion of fairness.

If we assume that the guards are selected in “fair random order,” then the program given at the beginning of this section always terminates and can set x to any natural number. Here fairness refers to *weak fairness*, or *justice*, which requires that each guard that continuously evaluates to true is eventually chosen. So the assumption of weak fairness for guarded commands allows us to implement the program $x :=?$.

As shown in Boom [1982] and independently, though later, in Apt and Olderog [1983], the converse holds as well. We follow here a presentation from the latter paper. We call a guarded commands program *deterministic* if in each alternative or repetitive command used in it the guards are mutually exclusive. We call a guarded commands program

$$S \equiv S_0; \textbf{do} \ \square_{i=1}^{n} B_i \longrightarrow S_i \ \textbf{od},$$

one-level nondeterministic if $S_0, S_1, \ldots, S_n$ are deterministic programs.

Given now such a one-level nondeterministic program, we transform it into the following guarded commands program that uses the $x :=?$ programs:

```
$T_{wf}(S) \equiv S_0;\ z_1 :=?;\ \ldots;\ z_n :=?;$
            do  $\square_{i=1}^{n} B_i \wedge z_i = min\,\{z_k \mid 1 \le k \le n\} \longrightarrow$
                    $z_i :=?;$
                    for all $j \in \{1, \ldots, n\} \setminus \{i\}$ do
                        if $B_j$ then $z_j := z_j - 1$ else $z_j :=?$ fi
                    od;
                    $S_i$
            od,
```

where $z_1, \ldots, z_n$ are integer variables that do not occur in S.

Intuitively, the variables $z_1, \ldots, z_n$ represent priorities assigned to the n guarded commands in the repetitive command of S. A guarded command i has higher priority than a command j if $z_i < z_j$. Call a guarded command *enabled* if its guard evaluates to true.

Initially, the commands are assigned arbitrary priorities. During each iteration of the transformed repetitive command an enabled command with the maximal priority, that is, with the minimum value of z_i, is selected. Subsequently, its priority gets reset arbitrarily, while the priorities of other commands are appropriately modified: if the command is enabled then its priority gets increased and otherwise it gets reset arbitrarily. The idea is that by repeatedly increasing the priority of a continuously enabled guarded command we ensure that it will be eventually selected.

This way the transformation models weak fairness. More precisely, if we ignore the values of the variables $z_1, \dots, z_n$, the computations of $T_{wf}(S)$ are exactly the weakly fair computations of S. A similar transformation can be shown to model a more demanding form of fairness (called *strong fairness* or *compassion*) according to which each guard that infinitely often evaluates to true is also infinitely often selected. An interested reader can consult Apt and Olderog [1983] or chapter 12 of Apt et al. [2009].

But are there some interesting guarded command programs for which the assumption of fairness is of relevance? The answer is "yes." A nice example was provided to us some time ago by Patrick Cousot who pointed out that a crucial algorithm in their landmark paper [Cousot and Cousot 1977], which introduced the idea of abstract interpretation, relies on fairness. The authors were interested in computing a least fixed point of a monotonic operator in an asynchronous way by means of so-called chaotic iterations.

Recall that a *partial order* is a pair $(A, \sqsubseteq)$ consisting of a set A and a reflexive, antisymmetric and transitive relation $\sqsubseteq$ on A. Consider the n-fold Cartesian product A^n of A for some $n \geq 2$ and extend the relation $\sqsubseteq$ componentwise from A to A^n. Then $(A^n, \sqsubseteq)$ is a partial order.

Next, consider a function

$$F : A^n \longrightarrow A^n,$$

and the ith component functions $F_i : A^n \longrightarrow A$, where $i \in \{1, \dots, n\}$, each defined by

$$F_i(\bar{x}) = y_i \text{ iff } F(\bar{x}) = \bar{y}.$$

Suppose now that F is *monotonic*, that is, whenever $\bar{x} \sqsubseteq \bar{y}$ then $F(\bar{x}) \sqsubseteq F(\bar{y})$. Then the functions F_i are monotonic as well.

Further, assume that A^n is finite and has the $\sqsubseteq$-least element that we denote by $\emptyset$. By the Knaster and Tarski Theorem [Tarski 1955] F has a $\sqsubseteq$*-least fixed point* $\mu F \in A^n$. As in Cousot and Cousot [1977], we wish to compute μF *asynchronously*. This is achieved by means the following guarded commands program:

```
$\bar{x} := \emptyset;$
$\textbf{do} \; []_{i=1}^{n} \; \bar{x} \neq F(\bar{x}) \longrightarrow x_i := F_i(\bar{x}) \; \textbf{od}$
```

This program can diverge, but it always terminates under the assumption of weak fairness. This is a consequence of a more general theorem proved in Cousot and Cousot [1977]. An assertional proof of correctness of this program under the fairness assumption is given in Apt et al. [2009].

Dijkstra had an ambiguous attitude to fairness. As noted by Allen Emerson in his Chapter 23, Dijkstra stated in his EWD310, which appeared as Dijkstra [1971], that sequential processes forming a parallel program should "proceed with speed ratios, unknown but for the fact that the speed ratios would differ from zero" and referred to this property as "fairness."

In EWD391 dating from 1973 and published as Dijkstra [1982a], when introducing self-stabilization he wrote:

> In the middle of the ring stands a demon, each time giving, in "fair random order" one of the machines the command "to adjust itself". (In "fair random order" means that in each infinite sequence of successive commands issued by the d[a]emon, each machine has received the command to adjust itself infinitely often.)

In the two-page journal publication Dijkstra [1974] that soon followed, the qualification "fair random order" disappeared but "daemon" that ensures it remained:

> In order to model the undefined speed ratios of the various machines, we introduce a central daemon [...].

However, as we have seen when discussing the origin of guarded commands, he rejected in EWD399 [Dijkstra 1973b] fairness at the level of guarded commands. In EWD798 [Dijkstra 1981] one can find the following revealing comment:

> David Park (Warwick University) spoke as a last minute replacement for Dana Scott on "Fairness". The talk was well-prepared and carefully delivered, but I don't care very much for the topic.

Further, in his EWD1013 [Dijkstra 1987], titled "Position paper on 'fairness'," he plainly turned against fairness and ended his informal discussion by stating that "My conclusion [...] is that fairness, being an unworkable notion, can be ignored with impunity."

It is easy to check that the transformation given in Section 8.6 translates fairness for parallel programs assumed in Dijkstra [1971] to weak fairness for guarded commands. We are bound to conclude that Dijkstra's opinions on fairness were not consistent over the years.

At the time Dijkstra [1987] wrote his controversial note, fairness was an accepted and a well-studied concept, see, for example, Francez [1986]. The note did not change researchers' perception. It was soon criticized, in particular by Leslie Lamport and Fred Schneider [1988], who concluded that Dijkstra's arguments against fairness apply equally well against termination, or more generally, against any liveness notion.

8.9 Nondeterminism: Further Developments

Dijkstra's guarded commands language was not the last word on nondeterminism in computer science. Subsequent developments, to which he did not contribute, brought new insights, notably by clarifying the consequences of both angelic and demonic nondeterminism in the context of parallelism. In what follows we provide a short account of this subject.

8.9.1 Taxonomy of Nondeterminism

Among several papers dealing with nondeterminism in the wake of Dijkstra's guarded commands, we would like to single out two. Harel and Pratt [1978] investigated nondeterministic programs in the context of Dynamic Logic and related their work to the weakest precondition approach of Dijkstra. In their approach, the programs were built up from assignments to simple variables using a set of basic operators: $\cup$ for nondeterministic choice and ";" for sequential composition, $B?$ for testing a Boolean condition B, and * for iteration. An alternative command **if** $[]_{i=1}^{n} B_i \ \longrightarrow\ S_i$ **fi** can be viewed as an abbreviation for the program $\cup_{i=1}^{n}(B_i?; S_i)$ and a repetitive command **do** $[]_{i=1}^{n} B_i \ \longrightarrow\ S_i$ **od** an abbreviation for the program $((\bigvee_{i=1}^{n} B_i?); (\cup_{i=1}^{n}(B_i?; S_i)))^*; \neg(\bigvee_{i=1}^{n} B_i?)$, see, for example, de Bakker [1980].

Semantically, each program denotes a binary relation on states, augmented with the symbols $\perp$ representing *divergence* (nonterminating computations) and f representing *failure*, that is, a test evaluating to false without having any immediate alternative to pursue. Such a relation describes the input–output behavior of a given program. It is defined by induction on the structure of programs. For example, the relation associated with the program $S_1 \cup S_2$ is the union of the relations associated with the programs S_1 and S_2.

The input/output relation does not describe how it is computed in a step-by-step manner. When executed in a given state, the program $S_1 \cup S_2$ chooses either S_1 or S_2 to compute the successor state. For a given initial state, these nondeterministic choices can be systematically represented in a *computation tree*. Harel and Pratt distinguished four methods on how such a computation tree can be traversed: (1) depth first, (2) depth first with backtracking when a failure state in encountered, (3) breadth first, and (4) breadth first combined with ignoring failure states. For each method, they showed how to express the notion of total correctness in dynamic logic.

In the context of algebraic specifications of programming languages, Broy and Wirsing [1981] considered different kinds of nondeterminism in their paper. They called them: (1) *backtrack nondeterminism*, (2) *choice nondeterminism*, (3) *unbounded nondeterminism*, and (4) *loose nondeterminism*. Option (1) computes the whole set of possible outcomes, where any possibility of nontermination must be taken. The choice of the output comes "after" the computation of the set of all possible outputs. Option (2) corresponds to choices "during" the execution of alternative statements. Option (3) applies "prophetic" choices during the computation to avoid any nonterminating computations, thereby typically creating unboundedly many good outcomes. Finally, option (4) corresponds to choices "before" the execution of the program.

Broy and Wirsing were also early users of the terminology of *angelic* nondeterminism, which they identified with (3), *demonic* nondeterminism, which they identified with (1), and *erratic* nondeterminism, which they identified with (2). We could not trace who first introduced this terminology, though it has been often attributed to Tony Hoare.

8.9.2 Nondeterminism in a Context

In his books on CCS and the Π-calculus [Milner 1980, 1999], Robin Milner gave a simple example of two finite automata, one deterministic and one nondeterministic, that are equivalent when the accepted languages are compared. However, Milner argued that they are essentially different when they are considered as processes interacting with a user or an environment. The essence of the example is shown in Figure 8.3, adapted from Milner [1999]. Milner took this observation as a motivation to develop a new notion of equivalence between processes, called *bisimilarity* and based on the following notion of bisimulation, which is sensitive to nondeterminism.

Processes are like nondeterministic automata, with states and transitions between states that are labeled by action symbols. We write $p \xrightarrow{a} q$ for a transition from a state p to a state q labeled by a. A process has an initial state and may have infinitely many states and thus transitions.

A *bisimulation* between processes P and Q is a binary relation $\mathcal{R}$ between the states of P and Q such that whenever $p\mathcal{R}q$ holds, then every transition from p can be simulated by a transition from q with the same label, such that the successor states are again in the relation $\mathcal{R}$, and vice versa, every transition from q can be simulated by a transition from p with the same label such that the successor states are again in the relation $\mathcal{R}$. Processes are called *bisimilar* if there exists a bisimulation relating the initial states of the processes.

The processes shown in Figure 8.3 are *not* bisimilar. Indeed, suppose that $\mathcal{R}$ is a bisimulation with $p_1\mathcal{R}q_1$. Then the transition $p_1 \xrightarrow{i} p_2$ can be simulated only by $q_1 \xrightarrow{i} q_2$, which implies $p_2\mathcal{R}q_2$. However, now the transition $p_2 \xrightarrow{c} p_4$ cannot be simulated from q_2 because there is no transition with label c. Contradiction. This formalizes the intuition that only process P offers both tea and coffee, whereas Q offers either tea or coffee.

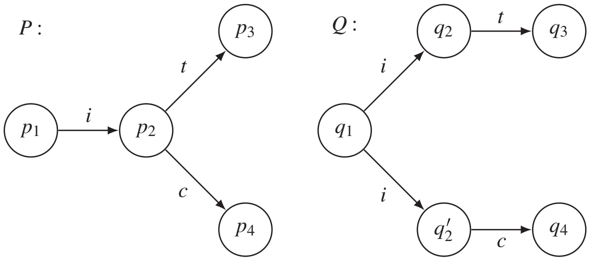


Figure 8.3 Two automata, P being deterministic and Q being nondeterministic on input of i, accept the same language consisting of the words *it* and *ic*. However, when viewed as processes interacting with a user, they are different. Suppose the process models a vending machine, i corresponds to the user's action of inserting a coin into the machine, and t and c to the user's choice of tea or coffee. Then, after insertion of the coin, P is in state p_2 and offers both tea and coffee to the user. However, Q makes a tacit choice by moving either to state q_2 or to state q_2' after a coin is inserted. In state q_2 it offers only tea, and in state q_2' only coffee, never both. Thus, from the user's perspective, the deterministic automaton is better because when using it no decision is taken without consulting her or him.

This new notion of equivalence triggered copious research activity resulting in various process equivalences that are sensitive to nondeterminism but differing in various other aspects, see, for example, van Glabbeek [2001]. Of particular interest is the idea of *testing* processes due to De Nicola and Hennessy [1984, 1987]. In these works, the interaction of a (nondeterministic) process and a user is explicitly formalized using a synchronous parallel composition. The user is formalized by a *test*, which is a process with some states marked as a *success*. For an example, see Figure 8.4. The authors distinguish between two options: a process may or must pass a test. A process P *may* pass a test T if in *some* maximal parallel computation with P, synchronizing on transitions with the same label, the test T reaches a *success* state. A process P *must* pass a test T if in *all* such computations the test T reaches a *success* state.

This leads to *may* and *must* equivalences. Two processes are *may* equivalent if each test that one process may pass the other may pass as well, and analogously for the *must* equivalence. So *may* equivalence corresponds to angelic nondeterminism, and *must* equivalence corresponds to demonic nondeterminism.

As an example, consider the processes P and Q from Figure 8.3 and the test T from Figure 8.4. Then P both may and must pass the test T, whereas Q only may pass it because in a synchronous parallel computation it can get stuck in the state pair (q_2, t_2) without being able to reach *success*. So in this simple example, both bisimilarity and *must* equivalence reveal the same difference between the deterministic process P and the nondeterministic process Q. In general, bisimilarity is finer than the testing equivalences, see again van Glabbeek [2001].

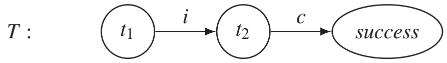


Figure 8.4 This test T checks whether a process can engage first in i and then c.

Also, CSP, originally built on Dijkstra's guarded commands as explained in Section 8.1, was developed further into a more algebraically oriented language that for clarity we call here "new CSP." It is described in Hoare's [1985] book. While guarded commands and the original CSP were notationally close to programming language constructs, where the nondeterminism appears only within the alternative command or the **do** loop, the new CSP introduced separate operators for each concept of the language. These can be freely combined to build up processes. We focus here on two nondeterministic operators introduced by Hoare.

Internal nondeterminism is denoted by the binary operator $\sqcap$, called *nondeterministic or* in Hoare [1985]. Informally, a process $P \sqcap Q$ "behaves like P or like Q, where the selection between them is made arbitrary, without knowledge or control of environment." In a formal operational semantics in the style of Plotkin [1980], this is modeled by using different labels of transitions. The special label τ appears at *internal* or *hidden* transitions, denoted by $p \xrightarrow{\tau} q$, which cannot be controlled or even seen by the environment. Labels $a \neq \tau$ appear at *external* or *visible* transitions, denoted by $p \xrightarrow{a} q$, and represent actions in which the environment can participate. The selection of $P \sqcap Q$ is modeled by the internal transitions $P \sqcap Q \xrightarrow{\tau} P$ and $P \sqcap Q \xrightarrow{\tau} Q$ [Roscoe 1998, 2010]. Thus, after this first hidden step, $P \sqcap Q$ behaves as P or as Q.

External nondeterminism or *alternation* is denoted by the binary operator $[]$, called *general choice* in Hoare [1985]. The idea of a process $P[]Q$ is that "the environment can control which of P and Q will be selected, provided that this control is exercised on the very first action." The formal operational semantics of the operator $[]$ in the style of Plotkin [1980] is more subtle than the one for $\sqcap$, see Olderog and Hoare [1986] and Roscoe [1998, 2010]. In applications, $P[]Q$ is performed in the context of a synchronous parallel composition with another process R modeling a user or an environment. Then the first visible transition with a label $a \neq \tau$ of R has to synchronize with a first visible transition $P[]Q$ with the same label a, thereby selecting P or Q of the alternative $P[]Q$. This formalizes Hoare's idea stated above.

These two nondeterministic operators have also been studied by De Nicola and Hennessy [1987] in the context of Milner's CCS. The authors write $\oplus$ for internal nondeterminism and keep $[]$ for external nondeterminism. When defining the operational semantics of the two operators, they write $\longrightarrow$ instead of $\xrightarrow{\tau}$ and speak of "CCS without τ's" because τ is not present in this process algebra.

Subsequently, they introduce a testing semantics and provide a complete algebraic characterization of the two operators.

To assess the effect of nondeterminism, the new CSP introduced a new equivalence between processes due to Brookes et al. [1984], called *failure equivalence*. A *failure* of a process is a pair consisting of a trace, that is, a finite sequence of symbols that label transitions, and a set of symbols that after performing the trace the process can *refuse*. Processes with the same set of failures are called *failure equivalent*. Besides an equivalence, new CSP also provides a notion of *refinement* among processes. A process P *refines* a process Q if the set of failures of P is a subset of the set of failures of Q. Informally, this means that P is *more deterministic* than Q. Thus, by definition, processes that refine each other are failure equivalent.

As an example, consider again the processes P and Q in Figure 8.3. They are not failure equivalent, but P refines Q. This example shows that failure equivalence is sensitive to nondeterminism. It turns out that failure equivalence coincides with the *must* equivalence for "strongly convergent" processes, that is, those without any divergences [De Nicola 1987]. So, it represents demonic nondeterminism.

8.10 Conclusions

As explained in this chapter, nondeterminism is a natural feature of various formalisms used in computer science. The proposals put forward prior to Dijkstra's [1975] seminal paper are examples of what is now called angelic nondeterminism. Dijkstra's novel approach, now called demonic nondeterminism, was clearly motivated by his prior work on concurrent programs that are inherently nondeterministic in their nature. His guarded command language became a simplest possible setting allowing one to study demonic nondeterminism, unbounded nondeterminism, and fairness.

Its versatility was demonstrated by subsequent works on diverse topics. In Martin [1986] correct delay-insensitive VLSI circuits were derived by means of a series of semantics-preserving transformation starting with a distributed programming language. In some aspects, the language is similar to CSP. In its sequential part it uses a subset of guarded commands with an appropriately customized semantics. To study randomized algorithms and their semantics, an extension of the guarded commands language with a probabilistic choice operator was investigated in a number of papers, starting with He et al. [1997]. More recently, guarded commands emerged in the area of quantum programming, as a basis for quantum programming languages, see, for example, Ying [2016].

As explained, the guarded commands language can also be viewed as a steppingstone towards a study of concurrent programs. In fact, it can be seen as a logical layer that lies between deterministic and concurrent programs.

The viability of Dijkstra's proposal can be best viewed by consulting statistics provided by Google Scholar. They reveal that the original paper, Dijkstra [1975], has been most often cited in the past decade.

Acknowledgements

We would like to thank Nachum Dershowitz, Peter van Emde Boas, and Reiner Hähnle for useful comments and Manfred Broy, Rob van Glabbeek, and Leslie Lamport for suggesting a number of helpful references.

References

H. Abelson and G. J. Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd. ed.). MIT Press.

K. R. Apt and E.-R. Olderog. 1983. Proof rules and transformations dealing with fairness. *Sci. Comput. Program*. 3, 65–100. DOI: https://doi.org/10.1016/0167-6423(83)90004-7.

K. R. Apt, L. Bougé, and P. Clermont. 1987. Two normal form theorems for CSP programs. *Inf. Process. Lett.* 26, 165–171. DOI: https://doi.org/10.1016/0020-0190(87)90001-9.

K. R. Apt, F. S. de Boer, and E.-R. Olderog. 2009. *Verification of Sequential and Concurrent Programs*, third, extended. Springer, New York. DOI: https://doi.org/10.1007/978-1-84882-745-5.

M. Armoni and M. Ben-Ari. 2009. The concept of nondeterminism: Its development and implications for teaching. *ACM SIGCSE Bull.* 41, 2, 141–160. DOI: https://doi.org/10.1145/1595453.1595495.

E. Ashcroft and Z. Manna. 1971. Formalization of properties of parallel programs. *Mach. Intell*. 6, 17–41.

R. Back. 1980. Semantics of unbounded nondeterminism. In *Proceedings of the 7th Colloquium Automata, Languages and Programming*, Vol. 85: Lecture Notes in Computer Science. Springer, 51–63. DOI: https://doi.org/10.1007/3-540-10003-2_59.

H. J. Boom. 1982. A weaker precondition for loops. *ACM Trans. Program. Lang. Syst.* 4, 4, 668–677. DOI: https://doi.org/10.1145/69622.357189.

S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. 1984. A theory of communicating sequential processes. *J. ACM* 31, 3, 560–599. DOI: https://doi.org/10.1145/828.833.

M. Broy and M. Wirsing. 1981. On the algebraic specification of nondeterministic programming languages. In E. Astesiano and C. Böhm (Eds.), *Proceedings of the 6th Colloquium Trees in Algebra and Programming (CAAP'81)*, Vol. 112: Lecture Notes in Computer Science. Springer, 162–179. ISBN 3-540-10828-9. DOI: https://doi.org/10.1007/3-540-10828-9_61.

A. K. Chandra. 1978. Computable nondeterministic functions. In *19th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 127–131. DOI: https://doi.org/10.1109/SFCS.1978.10.

N. Chomsky. June. 1959. On certain formal properties of grammars. *Inf. Control* 2, 2, 137–167. DOI: https://doi.org/10.1016/S0019-9958(59)90362-6.

J. Cohen. 1979. Non-deterministic algorithms. *ACM Comput. Surv.* 11, 2, 79–94. DOI: https://doi.org/10.1145/356770.356773.

A. Colmerauer and P. Roussel. 1996. The birth of Prolog. In T. J. Bergin and R. G. Gibson (Eds.), *History of Programming Languages II*. ACM, 331–367. DOI: https://doi.org/10.1145/234286.1057820.

S. A. Cook. 1971. The complexity of theorem-proving procedures. In M. A. Harrison, R. B. Banerji, and J. D. Ullman (Eds.), *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, May 3–5, 1971, Shaker Heights, OH. ACM, 151–158. DOI: https://doi.org/10.1145/800157.805047.

P. Cousot and R. Cousot. 1977. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *ACM Symposium on Artificial Intelligence and Programming Languages*. ACM SIGPLAN Not. 12, 8, 1–12. DOI: https://doi.org/10.1145/872734.806926.

M. D. Davis. 1958. *Computability and Unsolvability*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill.

J. W. de Bakker. 1980. *Mathematical Theory of Program Correctness*. Prentice-Hall International, Englewood Cliffs, NJ.

R. De Nicola. 1987. Extensional equivalences for transition systems. *Acta Inf.* 24, 2, 211–237. DOI: https://doi.org/10.1007/BF00264365.

R. De Nicola and M. Hennessy. 1984. Testing equivalences for processes. *Theor. Comput. Sci.* 34, 83–133. DOI: https://doi.org/10.1016/0304-3975(84)90113-0.

R. De Nicola and M. Hennessy. 1987. CCS without τ's. In H. Ehrig, R. A. Kowalski, G. Levi, and U. Montanari (Eds.), *TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'87)*, Vol. 249: Lecture Notes in Computer Science. Springer, 138–152. DOI: https://doi.org/10.1007/3-540-17660-8_53.

E. W. Dijkstra. 1968a. Go to statement considered harmful. *Commun. ACM* 11, 3, 147–148. DOI: https://doi.org/10.1145/362929.362947.

E. W. Dijkstra. 1968b. Cooperating sequential processes. In F. Genuys (Ed.), *Programming Languages: NATO Advanced Study Institute*. Academic Press Ltd., London, 43–112. Originally published as EWD123 in 1965.

E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inform.* 1, 115–138. DOI: https://doi.org/10.1007/BF00289519.

E. W. Dijkstra. 1973a. Sequencing primitives revisited. EWD398. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD398.PDF.

E. W. Dijkstra. 1973b. An immediate sequel to EWD398: "Sequencing primitives revisited.". EWD399. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD399.PDF.

E. W. Dijkstra. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11, 643–644. DOI: https://doi.org/10.1145/361179.361202. Originally published as EWD426 in 1974.

E. W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 453–457. DOI: https://doi.org/10.1145/360933.360975.

E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

E. W. Dijkstra. 1981. Trip report E.W. Dijkstra, Newcastle, 19–25 July 1981. EWD798. http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD798.PDF.

E. W. Dijkstra. 1982a. Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 41–46. Originally published as EWD391 in 1973. DOI: https://doi.org/10.1007/978-1-4612-5695-3_7.

E. W. Dijkstra. 1982b. On weak and strong termination. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 355–357. Originally published as EWD673 in 1978. DOI: https://doi.org/10.1007/978-1-4612-5695-3_64.

E. W. Dijkstra. 1982c. The equivalence of bounded nondeterminacy and continuity. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 358–359. Originally published as EWD675 in 1978. DOI: https://doi.org/10.1007/978-1-4612-5695-3_65.

E. W. Dijkstra. 1987. Position paper on "fairness." *Softw. Eng. Notes* 13, 2, 18–20. EWD1013. https://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1013.PDF. DOI: https://doi.org/10.1145/43846.43848.

L. Flon and N. Suzuki. 1978. Nondeterminism and the correctness of parallel programs. In E. J. Neuhold (Ed.), *Formal Description of Programming Concepts*. North-Holland, Amsterdam, 598–608.

L. Flon and N. Suzuki. 1981. The total correctness of parallel programs. *SIAM J. Comput.* 10, 2, 227–246. DOI: https://doi.org/10.1137/0210016.

R. W. Floyd. 1967. Nondeterministic algorithms. *J. ACM* 14, 4, 636–644. DOI: https://doi.org/10.1145/321420.321422.

N. Francez. 1986. *Fairness*. Springer, New York. DOI: https://doi.org/10.1007/978-1-4612-4886-6.

D. Gries. 1981. *The Science of Programming*. Springer, New York. DOI: https://doi.org/10.1007/978-1-4612-5983-1.

D. Harel and V. R. Pratt. 1978. Nondeterminism in logics of programs. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, 203–213. DOI: https://doi.org/10.1145/512760.512782.

J. He, K. Seidel, and A. McIver. 1997. Probabilistic models for the guarded command language. *Sci. Comput. Program.* 28, 2–3, 171–192. DOI: https://doi.org/10.1016/S0167-6423(96)00019-6.

M. Hennessy. 1988. *Algebraic Theory of Processes*. MIT Press series in the foundations of computing. MIT Press. ISBN 978-0-262-08171-9.

H. Hermes. 1965. *Enumerability, Decidability, Computability—An Introduction to the Theory of Recursive Functions*. Springer. DOI: https://doi.org/10.1007/978-3-662-11686-9.

C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8, 666–677. DOI: https://doi.org/10.1145/359576.359585.

C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall. ISBN 0-13-153271-5.

J. E. Hopcroft and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley. ISBN 0-201-02988-X.

D. König. 1927. Über eine Schlußweise aus dem Endlichen ins Unendliche. *Acta Litt. Ac. Sci.* 3, 121–130.

R. Kowalski. 1974. Predicate logic as a programming language. In *Proceedings IFIP'74*. North-Holland, 569–574.

L. Lamport and F. Schneider. 1988. Another position paper on fairness. *Softw. Eng. Notes* 13, 3, 1–2.

L. A. Levin. 1973. Universal sequential search problems. *Probl. Peredachi Inf.* 9, 3, 115–116. In Russian. http://www.mathnet.ru/links/6fd998c9343e02f4cec2527c2fe1314e/ppi914.pdf.

A. J. Martin. 1986. Compiling communicating processes into delay-insensitive VLSI circuits. *Distrib. Comput.* 1, 4, 226–234. DOI: https://doi.org/10.1007/BF01660034.

J. McCarthy. 1963. A basis for a mathematical theory of computation. In B. Bradfort and D. Hirschberg (Eds.), *Computer Programming and Formal Systems*. North-Holland, 33–70.

R. Milner. 1980. *A Calculus of Communicating Systems*, Vol. 92: Lecture Notes in Computer Science. Springer-Verlag, New York. DOI: https://doi.org/10.1007/3-540-10235-3.

R. Milner. 1999. *Communicating and Mobile Systems—The Pi-calculus*. Cambridge University Press. ISBN 978-0-521-65869-0.

E.-R. Olderog and C. A. R. Hoare. 1986. Specification-oriented semantics for communicating processes. *Acta Inf.* 23, 1, 9–66. DOI: https://doi.org/10.1007/BF00268075.

S. Owicki and D. Gries. 1976. An axiomatic proof technique for parallel programs. *Acta Inf.* 6, 4, 319–340. DOI: https://doi.org/10.1007/BF00268134.

G. D. Plotkin. 1976. A powerdomain construction. *SIAM J. Comput.* 5, 3, 452–487. DOI: https://doi.org/10.1137/0205035.

G. D. Plotkin. 1980. Dijkstra's predicate transformers & Smyth's power domains. In D. Bjørner (Ed.), *Abstract Software Specifications, 1979 Copenhagen Winter School, Proceedings*, Vol. 86: Lecture Notes in Computer Science. Springer, 527–553. DOI: https://doi.org/10.1007/3-540-10007-5_48.

M. O. Rabin and D. S. Scott. 1959. Finite automata and their decision problems. *IBM J. Res. Dev.* 3, 2, 114–125. DOI: https://doi.org/10.1147/rd.32.0114.

H. Rogers Jr. 1987. *Theory of Recursive Functions and Effective Computability (Reprint from 1967)*. MIT Press. ISBN 978-0-262-68052-3. https://mitpress.mit.edu/books/theory-recursive-functions-and-effective-computability.

A. W. Roscoe. 1998. *The Theory and Practice of Concurrency*. Prentice Hall.

A. W. Roscoe. 2010. *Understanding Concurrent Systems*. Texts in Computer Science. Springer. ISBN 978-1-84882-257-3. DOI: https://doi.org/10.1007/978-1-84882-258-0.

W. J. Savitch. 1970. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.* 4, 2, 177–192. DOI: https://doi.org/10.1016/S0022-0000(70)80006-X.

D. C. Smith and H. J. Enes. 1973. Backtracking in MLISP2: An efficient backtracking method for LISP. In *IJCAI'73*. William Kaufmann, 677–685.

E. Spaan, L. Torenvliet, and P. van Emde Boas. 1989. Nondeterminism, fairness and a fundamental analogy. *Bull. EATCS* 37, 186–193.

A. Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.* 5, 285–309.

Terese. 2003. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science 55. Cambridge University Press, Cambridge, UK.

A. M. Turing. 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc*. s2-42, 1, 230–265. DOI: https://doi.org/10.1112/plms/s2-42.1.230.

R. J. van Glabbeek. 2001. The linear time–branching time spectrum. I. In J. A. Bergstra, A. Ponse, and S. A. Smolka (Eds.), *Handbook of Process Algebra*. North-Holland/Elsevier, 3–99. ISBN 978-0-444-82830-9. DOI: https://doi.org/10.1016/b978-044482830-9/50019-9.

M. Ying. 2016. *Foundations of Quantum Programming*. Morgan Kaufmann.

R. Zabih, D. A. McAllester, and D. Chapman. 1987. Non-deterministic Lisp with dependency-directed backtracking. In *Proceedings of the 6th National Conference on Artificial Intelligence*. Morgan Kaufmann, 59–65.

D. Zöbel. 1988. Normalform-Transformationen für CSP-Programme. *Informatik: Forschung und Entwicklung* 3, 64–76.

9

A Personal View of Edsger W. Dijkstra and His Stance on Software Construction

Christian Lengauer

When I studied the book on structured programming [Dijkstra 1972] as a mathematics student at the Free University of Berlin in the mid-seventies, I could not have fathomed that I would ever interact with Edsger W. Dijkstra—let alone gain his friendship. But when I had joined UT Austin on an assistant professorship in the eighties, he joined two years later. He offered me a membership in his Austin Tuesday Afternoon Club (ATAC), and I also had the pleasure of experiencing the generous and sociable side of Edsger and his wife Ria.

Some time after they had gotten settled, I was invited for dinner at the Dijkstra home. It was my first time with them alone. Like many others, I was a bit in awe of Edsger. Dinner was delicious and the conversation was relaxed and turned to many topics, with pauses in between. Edsger had the knack of starting to speak just when you were ready to say something. Of course, I gave way but I had to get used to it.

I had just returned from an extended trip to Europe and recounted, among other things, innocently a flight from Saarbrücken to Berlin-Tempelhof in a Fokker F-27, the popular Dutch twin-propeller airplane from the fifties. What made the flight notable were the vibrations that I felt, especially since I sat right under the wing. I had to keep my coffee cup from walking across the table. After the touchdown in Berlin, when the engines had died down, a meek voice from the back of the plane broke the dead silence, asking his companion: “Lebste noch?” (“You still alive?”)

I thought that funny and entertaining, but Edsger remained silent and Ria responded quietly: “But it’s a good plane...” Both were clearly taken aback, I supposed, because the F-27 is a Dutch product. Later that evening, Edsger told me about his past and the beginnings of his career as a "computational engineer" in the Netherlands.

I departed for home one hour before midnight. While waiting for a slow traffic light to turn green, I replayed in my mind the things that had been said—and only then did I realize that Edsger had quietly worked into the conversation that he himself had been involved in the coding of *the wing stability equations for the Fokker F-27*! Apparently, he had chosen not to embarrass me with an outright réplique. I was touched by this show of tact toward me, a junior colleague whom he barely knew. One of his earlier EWD notes [Dijkstra 1976b], which I was not aware of at the time, recounts his time at Fokker.

Some months later, Edsger called me at home and announced that he and Ria would like to come over immediately. My alarm whether there was a serious reason was unfounded. They just wanted to spend time chatting and sharing a drink. It must have been at or after 9pm because, at that time of the day, Edsger gave himself permission for an alcoholic beverage. I had acquired recordings of the Vienna New Year concerts of the early eighties and played Strauss waltzes when they arrived. After an hour I switched to Haydn. Edsger inhaled cigarette smoke with pleasure and said quietly: "Ah, the music has improved."

He and I enjoyed exchanging classical music via self-recorded cassette tapes. He always wrote out the contents of his tape carefully on the cassette cover, with the recording date and a dedication to me. This way I learned that he hated the romantic variety, most of all Tchaikovsky, and was also not fond of waltzes or organ music. This realization came not before I had brought him an organ record from the Passau cathedral. He had accepted the gift without complaint.

Edsger and Ria showed me their interest and affection not only in Austin. Maybe there was a mutual kinship between us as Europeans in Texas. When I attended the conference PARLE'89 in Eindhoven, they insisted that I stay over at their home in Nuenen. Edsger, wearing his "Don't mess with Texas" t-shirt, took me on a tour on the backseat of his tandem bike (see Figure 9.1). It was a truly unique experience for me. Later, I discovered that he liked to treat his computer science guests to this ride [Dijkstra 1989].

Professionally, Edsger's impact on me is best summarized by Rule 0 of his Austin Tuesday Afternoon Club:[1] "Don't make a mess of it." It made me strive for simplicity in notation and modeling throughout my scientific career and take unpleasant complexity as an indication of a possible lack of comprehension on the part of the author. In my previous work at the University of Toronto, under the supervision of Rick Hehner, I had become familiar with predicate transformers

[1] Every member of the ATAC was given a coffee mug with this rule printed on it.

Figure 9.1 Edsger W. Dijkstra and Christian Lengauer. Photo by Ria Dijkstra in Nuenen, The Netherlands, June 1989.

[Dijkstra 1976a] and the view of programs as mathematical objects, and so I felt very much at home in Edsger's conceptual world.

* * *

The concept that pervaded my research more than any other is the *semantics-preserving program transformation* for the purpose of a program's optimization. One way of improving a program is to replace one piece of it with another that has the same input–output behavior but which is in some other sense superior.

A prerequisite is the ability to establish the semantic equivalence of programs, something that had not been—and still is not—at the forefront of the minds of imperative programmers. Equational reasoning is most easily accomplished if one gives up the very heart of imperative programming, the re-assignment, since it violates *referential transparency*, that is, the ability to replace equals with equals. Indeed, John Backus, in his Turing Award lecture in 1977 [Backus 1978], had given wide exposure to this idea.

Aside. Backus' Turing Award lecture incensed Edsger. He and Backus had a heated letter exchange over a couple of years that was only recently made public [Chen 2016]. Aside from Edsger's aversion to uncompromisingly formal deductions at the time, it seemed to him at too premature a stage for presentation in an ACM Turing Award lecture. My first contact with both of them was as a junior Ph.D. student at the Summer School on Programming Methodology in Santa Cruz in 1979. I happened to take a photograph of the two of them, sitting with others pleasantly at the same table. I had no idea what had transpired between them and just had come to a mutually agreeable end.

In the presence of the re-assignment, the equality of imperative programs can be established with the detour via a formal semantics. The semantics of two programs can be compared and, if they are equal, one program can take the place of the other anywhere in a series of code optimizations.

This reduces the risk of introducing errors in the optimization of sequential programs, but it is even more valuable when introducing parallelism. My dissertation work [Lengauer and Hehner 1982] pursued the idea by adorning an imperative sequential program with declarations of so-called *semantic relations* of program statements. These relations permit the replacement of sequential code with other sequential code or with interference-free parallel code.[2] This was in contrast to the widespread approach of composing sequential processes that interfere with each other unless one prevents the interferences with exclusions and synchronization. With the declaration of semantic relations, one gets from the sequential to the parallel program via a sequence of replacement steps all of which produce an executable program and maintain semantic equivalence. Alas, the dominant view of parallelism at the time was as a means for structuring programs, not for optimizing them, and our proposal did not change that.

At the ATAC in Austin, I was reassured to pursue the path of formal program synthesis as opposed to informal program derivation and post hoc formal verification. An early motivator was for me the new textbook by David Gries [1981] that

[2] The ultimate semantic relation, the independence of two program statements, was later completely formalized in universal algebra [Fränzle and Lengauer 2011].

the author called tongue-in-cheek “Dijkstra for the Masses.” Part III describes how to derive, in the weakest precondition calculus, loop termination conditions and invariants from a loop’s postcondition.

Aside. Another point of the book that got my attention already in 1978, when the original paper appeared [Gries 1978], and which pointed (for me) at things to come, was the functional view of the array as a monolithic variable, rather than the customary, imperative view as a collection of individual element variables in which a re-assignment represents an isolated update of an array element. An earlier EWD note [Dijkstra 1974] recommends the same view.

Edsger had some pet peeves that I found convincing. Like him, I preferred textual representations and remained suspicious of pictorial approaches to formality.[3] Pictures invite ambiguity far too easily [Dijkstra 1978]. An early case in point for me was the computational complexity of graph equality, that is, graph isomorphism, passionately portrayed in my graph theory course in Toronto by one of the problem’s great fans and contributors, Derek Corneil [Read and Corneil 1977]. Of course, Edsger did not reject the graph as a mathematical structure — his shortest-path algorithm deals with it. But, while pictures of graphs may suit as a means of illustration, they should not be used as the basis for a formal argument.

One other of Edsger’s pet peeves also convinced me readily: his disdain of unnecessary indices and function parameters. This was a turnabout from his rejection of the “point-free” combinator programming style one decade earlier, documented in the letter exchange with John Backus that I mentioned previously. Combinator expressions give rise to powerful equalities—for example, the fusion and homomorphism laws in Haskell [Bird 1998, Gorlatch 1999]—but, taken to the extreme, they can be difficult to understand. Early examples are the notorious APL one-liners. It seems that this semantic density is a challenge that comes with ultimate abstraction.

I fell in love with the language Haskell, which takes an extreme position by making referential transparency its highest priority and, as a consequence, banishing the re-assignment in any shape or form and taking the same view of arrays as Gries had taken a decade earlier. I was attuned to Haskell-style equational reasoning by Edsger’s strong preference for neatly formatted equational proofs [Dijkstra and Scholten 1990], which were dominant at the ATAC.

In parallelism as a means of increasing performance, I went on from manual to automatic transformation. This required a strictly formal, constructive deduction. Edsger preferred the manual analysis and synthesis and the proof by intuition.

[3] These were times at which “flawcharts,” as they were called around Edsger, were rampant.

He enjoyed the intellectual challenge and insisted on the unencumbered freedom of shaping the statement of the problem and its solution. Still, in his derivations, he often said: "There is really only one thing you can do...." This does hint at the potential for an automatic procedure, although he did not mean to do so.

Aside. The intuitive and the formal proof style are nicely contrasted by two alternative correctness proofs of a concurrent garbage collector for LISP programs: one by Edsger (and his coauthors) and the other by David Gries. Edsger [Dijkstra et al. 1978] poses and refines the necessary invariants for correct parallelism intuitively, while David [Gries 1977] works strictly with Hoare-style formulas.

In my research, to be able to derive an automatic procedure, I had to restrict the problem domain. I concentrated on the parallelization of highly iterative computations specified by recurrence equations or nested loops. Examples are matrix problems as used in scientific computing, image and signal processing, or simulation. The underlying data structure is the array, the computational platform a regular array of processes, in its simplest form called a *systolic array* [Kung and Leiserson 1979]. Regular process arrays were first realized in hardware, but with the advent of ever more powerful supercomputers, the domain was extended significantly and chiefly realized in software. If one requires linear affinity of the iteration bounds and array indices, the result is magic: you get a complete layout of the solution in time and space—the partial temporal order of all iteration steps, the spatial layout of the computations, and the distributions of all data streams. Integer linear programming even lets you perform an automatic search for an optimum solution by some chosen measure, for example, the minimal number of parallel steps on the minimal number of processors or with the maximum throughput.

This computational model for loop parallelization, the *polyhedron model* [Feautrier and Lengauer 2011], has been realized in a few widely used implementations, among them the LLVM plugin Polly [Grosser et al. 2012].[4] With a couple of seed papers in the sixties and seventies, the model had emerged in the eighties, and I turned to it in the early nineties, after having left Austin. I believe that Edsger was not aware of the polyhedron model. He would have approved of its mathematical rigor, but he would likely have abhorred the proliferation of indices that is customary in scientific computing particularly when working with arrays. Interestingly, there is an approach that de-emphasizes indices, called FLAME. It has actually been pursued by Robert van de Geijn [Gunnels et al. 2001], a departmental colleague of Edsger, but was started only around the time of Edsger's final departure from Austin.

[4] LLVM [Lattner and Adve 2004] is a popular collection of modular and reusable compiler and toolchain technologies.

A particularly ambitious branch of domain-specific program synthesis is that of *feature orientation* [Apel et al. 2010]. Here, programs are no longer viewed as individuals but as members of a family. The idea of a program family can be traced back to EWD249 [Dijkstra 1969] that subsequently became Edsger's chapter of the book on structured programming [Dijkstra 1972]. Half a decade later, David Parnas [1976] gave it exposure in the context of *program refinement*, that is, the stepwise concretization of a program from its specification to executable code.

In feature orientation, a *program family* is characterized by a set of concepts of the application domain, so-called *features*, which can be present or absent or which can take an instantiated, parametrized form in its programs. The family is also called a *software product line* [Apel et al. 2013]. A software tool composes—one says *weaves*—a program from its set of features, possibly applying optimizations that suit this specific composition. One widely used software product line is the Linux kernel whose number of features—like device drivers, network features, bus options, file systems, and so on—has grown to over 12,000 [Passos et al. 2021]. Admittedly, its implementation with the C preprocessor, cpp, is archaic by today's standards.

Edsger seemed almost ostentatiously disinterested in software tools—even in the computer per se—but reliable and effective tools are the ultimate consequence of Edsger's dream of making program construction and maintenance a mathematical discipline. Ideally, the tool embodies a powerful theory of some domain and protects the user from the burden of having to be versed in it. The view of a program as a mathematical object comes naturally in functional programming, but less so in imperative programming because, there, it requires the detour via a formal semantics. Still, I see this view gaining ground in imperative software engineering, embedded systems, and even in the high-pressure application sector of high-performance computing.

* *
*

Edsger was a person of strong opinions and convictions that he held passionately. This does not mean that he was generally unwilling or unable to change his mind. Above all, I observed an increasing formality in his theoretical work. The informal style of his early book [Dijkstra 1976a] is in clear contrast to his later book with Carel Scholten [Dijkstra and Scholten 1990], which is filled with equational proofs.

Edsger applied his convictions also to the community of computer scientists whom he liked to divide into good and bad. If he saw you on the bad side, he could be dismissive and downright mean. In one unfortunate such case, which I witnessed shortly after he arrived at UT Austin, this affected a junior's career.

If Edsger saw you on the good side, he could be gentle and lenient. This applied to me. Having had the opportunity of being around him regularly, I found it easy to avoid the traps of his dislikes. I have mentioned here some of his preferences that I adopted. But I could not be convinced to start numbering from zero [Dijkstra 1982] and, after a failed attempt, I decided against writing with a fountain pen. He never took issue with this, except in a letter that he posted seven weeks before his death: "I hope you will overcome your resistance and learn how to fill a pen without soiling your fingers, or otherwise you are denying yourself one of the joys of life." These were his last written words to me.

It was a true pleasure and an eye-opening privilege to have been part of Edsger's life.

References

S. Apel, C. Lengauer, B. Möller, and C. Kästner. 2010. An algebraic foundation for automatic feature-based program synthesis. *Sci. Comput. Program.* 75, 11, 1022–1047. DOI: https://doi.org/10.1016/j.scico.2010.02.001.

S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. DOI: https://doi.org/10.1007/978-3-642-37521-7.

J. Backus. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* 21, 8, 613–641. DOI: https://doi.org/10.1145/359576.359579.

R. Bird. 1998. *Introduction to Functional Programming Using Haskell* (2nd. edn.). Series in Computer Science. Prentice Hall.

J. Chen. 2016. This guy's arrogance takes your breath away (Letters between John W. Backus and Edsger W. Dijkstra, 1979). Medium. https://medium.com/@acidflask/this-guys-arrogance-takes-your-breath-away-5b903624ca5f.

E. W. Dijkstra. 1969. Notes on structured programming. EWD249. https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

E. W. Dijkstra. 1972. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.), *Structured Programming*, number 8 in A.P.I.C. Studies in Data Processing. Academic Press Ltd, 1–82.

E. W. Dijkstra. 1974. On the abolishment of the subscripted variable. EWD417. https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD417.PDF.

E. W. Dijkstra. 1976a. *A Discipline of Programming*. Series in Automatic Computation. Prentice-Hall.

E. W. Dijkstra. 1976b. A programmer's early memories. EWD568. https://www.cs.utexas.edu/users/EWD/ewd05xx/EWD568.PDF.

E. W. Dijkstra. 1978. Written in anger. EWD696. https://www.cs.utexas.edu/users/EWD/ewd06xx/EWD696.PDF.

E. W. Dijkstra. 1982. Why numbering should start at zero. EWD831. https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF.

E. W. Dijkstra. 1989. Andrej P. Ershov in Nuenen. EWD1057. https://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1057.PDF.

E. W. Dijkstra and C. S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag. DOI: https://doi.org/10.1007/978-1-4612-3228-5.

E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11, 966–975. DOI: https://doi.org/10.1145/359642.359655.

P. Feautrier and C. Lengauer. 2011. Polyhedron model. In D. Padua (Eds.), *Encyclopedia of Parallel Computing*. Springer-Verlag, 1581–1592. DOI: https://doi.org/10.1007/978-0-387-09766-4_502.

M. Fränzle and C. Lengauer. 2011. Semantic independence. In D. Padua (Eds.), *Encyclopedia of Parallel Computing*. Springer-Verlag, 1803–1810. DOI: https://doi.org/10.1007/978-0-387-09766-4_288.

S. Gorlatch. 1999. Extracting and implementing list homomorphisms in parallel program development. *Sci. Comput. Program.* 33, 1, 1–27. DOI: https://doi.org/10.1016/S0167-6423(97)00014-2.

D. Gries. 1977. An exercise in proving parallel programs correct. *Commun. ACM* 20, 12, 921–930. DOI: https://doi.org/10.1145/359897.359903.

D. Gries. 1978. The multiple assignment statement. *IEEE Trans. Softw. Eng.* SE-4, 2, 89–93. DOI: https://doi.org/10.1109/TSE.1978.231479.

D. Gries. 1981. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag. DOI: https://doi.org/10.1007/978-1-4612-5983-1.

T. Grosser, A. Größlinger, and C. Lengauer. 2012. Polly—Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 4, Article 1250010, 28. DOI: https://doi.org/10.1142/S0129626412500107.

J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. 2001. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Softw.* 27, 4, 422–455. DOI: https://doi.org/10.1145/504210.504213. https://github.com/flame/.

H. T. Kung and C. E. Leiserson. 1979. Algorithms for VLSI processor arrays. In C. Mead and L. Conway (Eds.), *Introduction to VLSI Systems*, Chapter 8. Addison-Wesley.

C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*. IEEE, 75–88. DOI: https://doi.org/10.1109/CGO.2004.1281665.

C. Lengauer and E. Hehner. 1982. A methodology for programming with concurrency: An informal presentation. *Sci. Comput. Program.* 2, 1, 1–18. DOI: https://doi.org/10.1016/0167-6423(82)90002-8.

D. L. Parnas. 1976. On the design and development of program families. *IEEE Trans. Softw. Eng.* 2, 1, 1–9. DOI: https://doi.org/10.1109/TSE.1976.233797.

L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. A. Padilla. 2021. A study of feature scattering in the Linux kernel. *IEEE Trans. Softw. Eng. (TSE)* 47, 1, 146–164. DOI: https://doi.org/10.1109/TSE.2018.2884911.

R. C. Read and D. G. Corneil. 1977. The graph isomorphism disease. *J. Graph Theory* 1, 4, 339–363. DOI: https://doi.org/10.1002/jgt.3190010410.

10

Applying Dijkstra’s Vision to Numerical Software

Robert van de Geijn and Maggie Myers

> Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer’s burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.—The Humble Programmer [Dijkstra 1972].

10.1 Introduction

In Spring 1987, one of us (Robert) interviewed for a faculty position in computer science at UT Austin. Part of his talk focused on a communication system for distributed memory computers, and he made the statement that it simplified debugging. A voice from the audience asserted that one should not have to debug. Having been trained as an applied mathematician, Robert did not consider this a serious criticism and hence he ignored it. After the talk, faculty congratulated him on how well he had handled the comment by Dijkstra. It took another decade for Dijkstra’s insights to become central to our[1] research.

In spite of also being Dutch, and for a while occupying adjacent offices, Dijkstra[2] and Robert did not interact much during their overlapping time at UT. After more

[1] In this paper, “we” and “our” refers to Robert, Maggie, and collaborators.

[2] We were never close enough with Edsger W. Dijkstra to comfortably use his first name here.

than a decade as colleagues, they went out with a visitor and the discussion turned to the recent winter in The Netherlands and the traditional "elfstedentocht," a skating event involving 11 cities. At some point, Dijkstra exclaimed in surprise something along the lines of "You are Dutch!" He then declared Robert the most amazing transformation of a Dutchman into an American he had encountered. Robert chose to interpret this as a compliment.

Starting in the 1990s, Robert was considered an expert on the parallelization of dense linear algebra (DLA) software. The process went roughly like this: someone decided that it was important to port some routine from LAPACK [Anderson et al. 1992] to a distributed memory computer. Sometimes it was straightforward and sometimes it was not. When it was not, some colleagues in the field might come and visit him. Robert would doodle on a chalkboard for a bit and propose an alternative algorithm for the same operation that parallelized well. They would write yet another paper. Rinse and repeat.

> Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.—EWD896 [Dijkstra 1984].

The "mistake" we made was that we started to think about what the process was by which we discovered new algorithms. Because Robert and Maggie were teaching "Analysis of Programs" using Gries' [1981] *The Science of Programming*, we realized that we started from the mathematical specification and *a priori* identified multiple loop invariants, from which we would then derive a family of algorithms. By choosing the algorithm that exhibited parallelism, we would solve the posed problem. In other words, we employed goal-oriented programming techniques as advocated by Dijkstra and his contemporaries to derive programs hand in hand with their proofs of correctness. This was a "mistake" in the sense that it exposed the process to everyone, transforming it from an art limited to the "high priests of high performance" to a science that could be mastered by all.

> It is time to unmask the computing community as a Secret Society for the Creation and Preservation of Artificial Complexity.—EWD1243a [Dijkstra 1996].

10.2 An Example

We will illustrate what we now call the Formal Linear Algebra Methods Environment (FLAME) methodology for deriving DLA algorithms with a simple example:

the inversion of an upper triangular matrix, U, overwriting the original matrix: $U := U^{-1}$. Letting $\widehat{U}$ denote the original contents of U and partitioning this matrix into quadrants, it can be easily verified [Bientinesi et al. 2008] that upon completion

$$\underbrace{\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)}_{U} = \underbrace{\left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right)}_{\widehat{U}^{-1}}, \tag{10.1}$$

where TL, TR, and so on should be read as “Top-Left,” “Top-Right,” and so on. (Square) submatrices U_{TL} and U_{BR} are themselves upper triangular matrices. This exposes a recursive definition of the inverse of an upper triangular matrix, which we call the *Partitioned Matrix Expression* (PME), for this operation.

10.2.1 The Goal

A standard technique for achieving high performance with a DLA algorithm is to iterate (loop) through matrices by blocks of rows and/or columns. Such “blocked algorithms” can achieve high performance on modern processors with complex memory hierarchies by casting most computation in terms of matrix–matrix multiplications. This means our goal is to transform the PME into a family of blocked algorithms, from which a member that (for example) parallelizes well can be chosen.

10.2.2 Notation, Notation, Notation

> How do we convince people that in programming simplicity and clarity—in short: what mathematicians call “elegance”—are not a dispensable luxury, but a crucial matter that decides between success and failure?—EWD648 [Dijkstra 1982].

As Dijkstra often advocated, it is important to employ context-appropriate notations. In our case, this means avoiding intricate indexing and expressing our algorithms in terms of simpler linear algebra operations so that only one loop is exposed in an algorithm.

Equation (10.1) can easily be translated into the algorithm in Figure 10.1, which we present there with what we now call the FLAME notation. The thick and thin lines have semantic meaning and indicate that, in the loop, quadrants of the matrix are repartitioned, submatrices are updated, and submatrices are added or subtracted to or from the quadrants. The algorithm requires the inversion of a smaller

```
Algorithm: $U := \textsc{Uinv\_blk\_var1}(U)$
$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$ where $U_{TL}$ is $0 \times 0$
while $m(U_{TL}) < m(U)$ do
    Determine block size $b$
    $\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$ where $U_{11}$ is $b \times b$

      $U_{11} := U_{11}^{-1}$
      $U_{01} := -U_{00} U_{01} U_{11}$

    $\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$
endwhile
```

Figure 10.1 A simple blocked algorithm for overwriting U with its inverse. Here, $m(A)$ equals the row size of matrix A.

$b \times b$ submatrix, U_{11}, which is typically accomplished via an "unblocked algorithm" that chooses the block size, b, equal to one so that the inversion of that submatrix (a scalar in this case) becomes the trivial inversion of a number.

Importantly, this notation allows the algorithm to be annotated with its proof of correctness as illustrated in Figure 10.2, where the assertions in the shaded boxes capture the state of the variables. In that figure, $\widehat{U}$ refers to the original contents of U and the loop invariant is given by

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right),$$

which captures that, so far, the "Top-Left" quadrant has been inverted. For conciseness, some information is implicit (e.g., the fact that U is itself upper triangular and that therefore other submatrices of U have special properties).

10.2.3 Deriving Algorithms

The problem with this simple algorithm, which follows directly from (10.1), is that it casts most computation in terms of multiplication with the upper triangular matrix U_{00}, which makes it hard to achieve high performance on a parallel architecture [Bientinesi et al. 2008]. What is needed is a mechanism to identify multiple algorithms in the hope of discovering one that parallelizes well. We do so by systematically identifying multiple loop invariants, from which algorithms can then be systematically derived.

<table>
<tr><th>Step</th><th>Algorithm: $U := \textsc{Uinv\_blk\_var1}(U)$</th></tr>
<tr><td>1a</td><td>$\left\{ U = \widehat{U} \right\}$</td></tr>
<tr><td>4</td><td>$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$ <b>where</b> U_{TL} is 0×0</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \right\}$</td></tr>
<tr><td>3</td><td><b>while</b> $m(U_{TL}) < m(U)$ <b>do</b></td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \wedge m(U_{TL}) < m(U) \right\}$</td></tr>
<tr><td>5a</td><td><b>Determine block size</b> b
$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$ <b>where</b> U_{11} is $b \times b$</td></tr>
<tr><td>6</td><td>$\left\{ \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \widehat{U}_{00}^{-1} & \widehat{U}_{01} & \widehat{U}_{02} \\ \hline 0 & \widehat{U}_{11} & \widehat{U}_{12} \\ \hline 0 & 0 & \widehat{U}_{22} \end{array}\right) \right\}$</td></tr>
<tr><td>8</td><td>$U_{11} := U_{11}^{-1}$
$U_{01} := -U_{00}U_{01}U_{11}$</td></tr>
<tr><td>7</td><td>$\left\{ \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \widehat{U}_{00}^{-1} & -\widehat{U}_{00}^{-1}\widehat{U}_{01}\widehat{U}_{11}^{-1} & \widehat{U}_{02} \\ \hline 0 & \widehat{U}_{11}^{-1} & \widehat{U}_{12} \\ \hline 0 & 0 & \widehat{U}_{22} \end{array}\right) \right\}$</td></tr>
<tr><td>5b</td><td>$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \right\}$</td></tr>
<tr><td></td><td><b>endwhile</b></td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \wedge \neg(m(U_{TL}) < m(U)) \right\}$</td></tr>
<tr><td>1b</td><td>$\left\{ U = \widehat{U}^{-1} \right\}$</td></tr>
</table>

Figure 10.2 A simple blocked algorithm for overwriting U with its inverse, annotated with its proof of correctness.

We start with the PME given in (10.1):

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right).$$

While a loop has not completed, only part of the final result has been computed. Hence, viable loop invariants can be constructed by examining partial results exposed in the PME:

$$\begin{array}{c|c|c}
\text{Invariant 1} & \text{Invariant 2} & \text{Invariant 3} \\
\left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) &
\left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) &
\left(\begin{array}{c|c} \widehat{U}_{TL}^{-1} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR} \end{array}\right) \\
\hline
\text{Invariant 4} & \text{Invariant 5} & \text{Invariant 6} \\
\left(\begin{array}{c|c} \widehat{U}_{TL} & \widehat{U}_{TR} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right) &
\left(\begin{array}{c|c} \widehat{U}_{TL} & -\widehat{U}_{TL}^{-1}\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right) &
\left(\begin{array}{c|c} \widehat{U}_{TL} & -\widehat{U}_{TR}\widehat{U}_{BR}^{-1} \\ \hline 0 & \widehat{U}_{BR}^{-1} \end{array}\right)
\end{array}.$$

This insight is a crucial piece for Dijkstra's vision of deriving correct algorithms: determining loop invariants *a priori*.

The annotated algorithm in Figure 10.2 now becomes a "worksheet," to be filled out in the order indicated by the numbers in the column labeled "Step." First, we enter the precondition and postcondition (Steps 1a and 1b). Then, we derive a PME (there could be more than one) and corresponding loop invariants, entering a chosen invariant in the worksheet where it must hold *true* (Step 2, in four places). By examining the loop invariant and postcondition, a loop guard (Step 3) is prescribed. The precondition and the loop invariant prescribe the initialization (Step 4). Because progress toward completion must be made, how to repartition (Steps 5a and 5b) is prescribed. Determining the state of the exposed submatrices after repartitioning is a matter of textual substitution and the application of linear algebra rules (Step 6). Knowing that the loop invariant must again hold at the bottom of the loop tells us what new state the exposed submatrices must take on (Step 7), which the reader may recognize as computing the weakest precondition. Comparing Steps 6 and 7 prescribes the updates that must happen (Step 8). Thus, we derive the algorithm hand in hand with its proof of correctness, as Dijkstra advocated.

We give the algorithmic variants that result from applying this process to Invariants 1–3, in Figure 10.3. Invariants 4–6 give rise to algorithms that sweep through the matrix in the opposite direction. Related details are given in Bientinesi et al. [2008].

The question we are left with is which algorithmic variant to choose. Variant 2 casts most of its computation in terms of the matrix–matrix multiplication $U_{02} := U_{02} + U_{01}U_{12}$. If U is $n \times n$ and U_{TL} is $k \times k$ then U_{02} is $(n-k-b) \times (n-k-b)$, U_{01} is $(n-k-b) \times b$, and U_{12} is $b \times (n-k-b)$. This is a shape of matrix–matrix multiplication that can attain high performance on a single core, multiple cores, and distributed memory architectures (provided n is large enough), see Bientinesi et al. [2008].

While an unblocked version of Variant 2 (discovered through alternative means) was included in LINPACK [Dongarra et al. 1979], this variant originally did not

```
Algorithm: $U := \textsc{Uinv\_blk\_var}1(U)$
$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$ where $U_{TL}$ is $0 \times 0$
while $m(U_{TL}) < m(U)$ do
    Determine block size $b$
    $\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{array}\right)$ where $U_{11}$ is $b \times b$
```

Variant 1	Variant 2	Variant 3
$U_{01} := -U_{00}U_{01}$		
	$U_{12} := -U_{11}^{-1}U_{12}$	$U_{12} := -U_{11}^{-1}U_{12}$
	$U_{02} := U_{02} + U_{01}U_{12}$	
$U_{01} := U_{01}U_{11}^{-1}$	$U_{01} := U_{01}U_{11}^{-1}$	
		$U_{12} := U_{12}U_{22}^{-1}$
$U_{11} := U_{11}^{-1}$	$U_{11} := U_{11}^{-1}$	$U_{11} := U_{11}^{-1}$

```
    $\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$
endwhile
```

Figure 10.3 Three algorithmic variants corresponding to Invariants 1–3. Computations such as $U_{12} := -U_{11}^{-1}U_{12}$ are actually implemented by solving the triangular system with multiple right-hand sides $U_{11}X = -U_{12}$, overwriting U_{12} with X, since this is numerically more stable and less compute intensive than first computing $U_{11} := U_{11}^{-1}$ and then multiplying with the result [Bientinesi et al. 2008].

appear in its successors, LAPACK [Anderson et al. 1992] and ScaLAPACK [Choi et al. 1992], which strove for high performance. Thus, the described methodology yielded an important new blocked algorithm for inverting a triangular matrix, as it did for many other DLA operations (see, e.g., Bientinesi et al. [2013]).

10.2.4 Representing Algorithms in Code

We have discussed the FLAME notation for representing DLA algorithms and the FLAME methodology for deriving them. When translating these derived-to-be-correct algorithms into code, we use Application Programming Interfaces (APIs) that allow the implementation to closely mirror the algorithm so that correctness transfers [Gunnels et al. 2001, Bientinesi et al. 2005b].

10.3 Decades of Research, Development, and Impact

The illustrated techniques build on the pioneering work of Dijkstra and his contemporaries in the 1960s and early 1970s. We now discuss how this has impacted numerical algorithms and the architecture of related software libraries.

10.3.1 Parallel Computing: Driving a Desperate Need for Simplicity

> [. . .] as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.—The Humble Programmer [Dijkstra 1972].

The need for hiding intricate indexing in software for DLA became particularly urgent with the advent of distributed memory architectures. Not only did a program need to track with what parts of a matrix to compute, but also on what node of the parallel computer what part of a matrix resided. This necessitated APIs for the C programming language that somewhat resembled the FLAME notation, as was part of the PLAPACK DLA library [Alpatov et al. 1997, van de Geijn 1997] in the mid-1990s.

10.3.2 Notation, Again

The first journal paper that used the FLAME notation presented a parallel variant on the Gauss–Jordan algorithm with pivoting for inverting a matrix [Quintana et al. 2001]. Although FLAME was not discussed in that paper and was not yet formalized as a method, the algorithm was derived by taking the classical approach that proceeds in three stages (LU decomposition with pivoting followed by inversion of U followed by solving $LX = U^{-1}$), deriving multiple algorithms for each stage, and merging the loops for appropriate algorithms into one that sweeps through the matrix only once. Interestingly, a request from a referee to discuss the numerical stability of the resulting algorithm was satisfied by arguing that the new algorithm merely merged the three stages of the traditional approach and hence inherited the numerical properties. Thus, this work prefigured future development.

10.3.3 FLAME

> First, one can remark that I have not done much more than to make explicit what the sure and competent programmer has already done for years, be it mostly intuitively and unconsciously. I admit so, but without any shame: making his behaviour conscious and explicit seems a relevant step in the process of transforming the Art of Programming into the Science of Programming. My point is that this reasoning can and should be done explicitly.—EWD209 [Dijkstra 1968].

The science behind our discovery of algorithms, which we dubbed the FLAME methodology, was first presented in a talk at the IFIP TC2/WG2.5 Working

Conference on the Architecture of Scientific Software [Gunnels and van de Geijn 2001], subsequently appeared in the ACM Transactions on Mathematical Software (TOMS) [Gunnels et al. 2001], and was part of a first dissertation related to FLAME by John Gunnels [Gunnels 2001]. This work gave an early overview of the notation for presenting algorithms, the methodology for deriving them, and the APIs for representing them in code.

10.3.4 Turning Knowledge into a System

> If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance. By definition this approach is only applicable when we restrict ourselves to intellectually manageable programs, but it provides us with effective means for finding a satisfactory one among these.—The Humble Programmer [Dijkstra 1972].

The FLAME methodology captured how to turn a specification of a DLA operation (the PME) into loop invariants and a loop invariant into a loop. Further progress came from formulating these steps as the "worksheet," illustrated in Figure 10.2, which made the process more explicitly systematic [Bientinesi et al. 2005a]. We used this to teach the methodology to undergraduates with limited linear algebra background. To their delight, deriving algorithms and translating them into code with the FLAME APIs yielded implementations that often gave the right answer the first time, despite the student not fully grasping what was being computed.

10.3.5 Making a System Mechanical

Once the methodology became a worksheet, it became obvious that the derivation of algorithms itself could be mechanized. As part of his Ph.D. dissertation [Bientinesi 2006], Paolo Bientinesi demonstrated this with a Mathematica implementation that took loop invariants as input and generated worksheets for algorithms, typeset similar to the algorithm in Figure 10.2. His research group subsequently perfected these techniques into a tool, Cl1ck, that starts with the specification of what is to be computed, produces one or more PMEs, derives loop invariants, and eventually outputs code in a choice of languages [Fabregat-Traver and Bientinesi 2011a, 2011b, Fabregat-Traver 2014].

10.3.6 Correctness in the Presence of Round-off Error

Correctness of a program takes on a different meaning when floating point arithmetic is employed. Generally, a numerical program is said to be correct (numerically stable) if it computes in floating point arithmetic the exact solution of a nearby problem. The idea is that the introduced error is indistinguishable from what results from a small error in the input data. This is known as the *backward error* and the analysis that bounds this error is known as a *backward error analysis*.

We have shown that backward error analyses for the algorithms that result from the FLAME process can themselves be derived via a goal-oriented approach [Bientinesi 2006, Bientinesi and van de Geijn 2011].

10.3.7 Sidestepping the Phase Ordering Problem

Given that the process generates a family of algorithms, a question arises on how to decide which algorithm to use when. In Tze Meng Low's [2013] dissertation, it is shown how a desirable property of an algorithm can be recognized from the relationship between the loop invariant and the PME from which it was obtained. For example, it can be determined whether loop reversal can be applied (an intermediate step in some optimizations), whether the loop can be easily checkpointed for fault tolerance, whether the loop parallelizes, or whether multiple loops can be merged [Low et al. 2005]. This, in some sense, sidesteps the phase-ordering problem encountered in compiler optimization.

In that dissertation, the relation between the FLAME methodology and primitive recursive functions is also explored.

10.3.8 Beyond Dense Linear Algebra

Had someone predicted 20 years ago that formal derivation would revolutionize the development of high-performance DLA software, we would have been skeptical. Now that we have demonstrated just that, the next question is to what other important domains these insights apply.

With our colleagues Victor Eijkhout and Paolo Bientinesi, we have successfully applied the methodology to the derivation of so-called Krylov subspace methods for solving linear systems involving sparse matrices [Eijkhout et al. 2010]. Key is the insight pioneered by Alsten Householder to assemble vectors from different iterations into the columns of matrices. This transforms the problem into the DLA domain.

Tze Meng Low and collaborators have similarly exploited the fact that many graph operations can be expressed using linear algebra operations. This has allowed them to apply the described techniques, systematically discovering high-performance algorithms for that domain [Lee and Low 2017, Low et al. 2017, 2018].

10.3.9 Turning Theory into Practice: Libflame

> If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.—The Humble Programmer [Dijkstra 1972].

Our thesis was that fundamentally the FLAME methodology, in conjunction with the FLAME APIs, provided a better way for developing DLA software libraries. To test this, we developed a new DLA library, libflame [Van Zee et al. 2009], with functionality that significantly overlapped with LAPACK.[3] For many of the most-used operations, families of algorithms were derived and implemented so that the best member for a given situation could be employed.

Over years of development, mostly by the primary developer of libflame (Field Van Zee), hundreds of algorithms were derived and implemented. Notably, no test suite was created until around five years into the project when a disruptive change to low-level routines in the library made this a prudent investment. The first time the test suite was run, tens of thousands of tests yielded a mere handful of errors, exactly in the low-level routines that had changed. These were easily fixed.

Around 2008, a gift from Microsoft encouraged us to take libflame to the next level, where it might actually attract users. This was subsequently further funded by grants from NSF's Software Infrastructure for Sustained Innovation (SI2) program. The resulting library is now distributed under an open source license, is part of the AMD Optimizing CPU Libraries (AOCL),[4] and is being embraced by Oracle in an effort to provide a high-performance solution for Java targeting machine-learning applications. Despite now having a very large user base, bug reports (which Dijkstra would have called errors) have been extremely rare.

10.4 Educating the Masses

> Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it is so extremely rewarding.—EWD1036 [Dijkstra 1988].

In the late 1990s, we started offering an undergraduate special topics course at UT Austin in which students learned how to systematically derive algorithms using

[3] LAPACK is reported to consist of millions of lines of code.

[4] https://developer.amd.com/amd-aocl/.

the FLAME methodology and the discussed worksheet. Participants were amazed to find out that they could easily discover new algorithms from the specification of a nontrivial linear algebra operation. Typically, their implementation computed the correct answer the first time they ran it. Years later, many recalled this as a transformative experience in their computer science education.

To share the practical importance of our use of formal methods for programming with the world, we have developed a Massive Open Online Course (MOOC) titled "LAFF-On Programming for Correctness," offered on the edX platform.[5] This course consists of two parts: The first part reviews the basics of logic needed to reason about programs, including Hoare logic and, in Dijkstra spirit, how to identify loop invariants *a priori*. This culminates in a worksheet similar to that given in Figure 10.2 but without the FLAME notation so that indices are still explicitly exposed. Only after the learners have mastered these tools do they finally derive and implement their first program. The second part introduces the FLAME notation so that the power of abstraction, and deriving algorithms hand in hand with their proofs of correctness, is fully experienced. A straightforward translation into code then yields a correct implementation that requires no testing. Thus, learners master and apply some of the mathematics that underlies the discipline of programming.

10.5 Conclusion

> In view of the well-known advice "Prevention is better than cure" not a surprising conclusion; yet it was a conclusion with considerable effects.—Programming methodologies, their objectives and their nature. EWD469 [Dijkstra 1976].

It is not that we set out to make some of Dijkstra's vision a reality in our area of expertise when we embarked on our journey two decades ago. Instead, as we analyzed how we discovered algorithms, we recognized we were, initially implicitly and eventually explicitly, applying and refining his techniques in formal methods. Dijkstra and his contemporaries were right: with appropriate abstraction and notation, programming can be a constructive endeavor that yields a proven-correct implementation. We would like to think that our work demonstrates this convincingly, in part because in our field there are clear measures of goodness: readability, robustness, portability, and the ultimate (parallel) performance of the resulting software. We believe that Dijkstra would have approved.

[5] https://www.edx.org/course/laff-on-programming-for-correctness, http://ulaff.net.

Acknowledgments

The efforts described in this paper involved a large number of collaborators, including members of the FLAME group (now called the Science of High-Performance Computing group) at UT Austin and elsewhere, most of whom are coauthors of the cited papers. We additionally thank Dr. Tim Mattson of Intel and Dr. Laurent Visconti from Microsoft for being patrons of this work at critical moments.

This material is based upon work supported by the National Science Foundation Grant Nos. ACI-0305163, CCF-0342369, CCF-0850750, CCF-0917096, ACI-1148125, and ACI-1550493. *Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.*

Additional support came from various industrial sources, most notably Intel, MathWorks, Microsoft, and NEC.

References

P. Alpatov, G. Baker, H. Edwards, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. 1997. PLAPACK Parallel Linear Algebra Package design overview. In *SC'97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*. IEEE, 29–29. DOI: https://doi.org/10.1109/SC.1997.10004.

E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. 1992. *LAPACK Users' Guide*. SIAM, Philadelphia.

P. Bientinesi. 2006. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. Ph.D. thesis. Department of Computer Sciences, The University of Texas. Technical Report TR-06-46.

P. Bientinesi and R. A. van de Geijn. March. 2011. Goal-oriented and modular stability analysis. *SIAM J. Matrix Anal. Appl.* 32, 1, 286–308. ISSN 0895-4798. DOI: https://doi.org/10.1137/080741057.

P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ort, and R. A. van de Geijn. 2005a. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw.* 31, 1, 1–26. http://doi.acm.org/10.1145/1055531.1055532.

P. Bientinesi, E. S. Quintana-Ort, and R. A. van de Geijn. 2005b. Representing linear algebra algorithms in code: The FLAME application program interfaces. *ACM Trans. Math. Softw.* 31, 1, 27–59. ISSN 0098-3500. DOI: https://doi.org/10.1145/1055531.1055533.

P. Bientinesi, B. Gunter, and R. A. van de Geijn. 2008. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.* 35, 1, 1–22. ISSN 0098-3500. DOI: https://doi.org/10.1145/1377603.1377606.

P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, T. Rhodes, R. A. van de Geijn, and F. G. Van Zee. 2013. Deriving dense linear algebra libraries. *Form. Asp. Comput*. 25, 6, 933–945. ISSN 1433-299X. DOI: https://doi.org/10.1007/s00165-011-0221-4.

J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 120–127. DOI: https://doi.ieeecomputersociety.org/10.1109/FMPC.1992.234898.

E. W. Dijkstra. 1968. A constructive approach to the problem of program correctness. *BIT Numer. Math.* 8, 174–186. DOI: https://doi.org/10.1007/BF01933419.

E. W. Dijkstra. 1972. The humble programmer. *Commun. ACM* 15, 10, 859–866. Turing Award lecture. DOI: https://doi.org/10.1145/355604.361591.

E. W. Dijkstra. 1976. Programming methodologies, their objectives and their nature. In *Structured Programming (Infotech State of the Art Report)*. Infotech International, 203–216.

E. W. Dijkstra. 1982. "Why is software so expensive?" An explanation to the hardware designer. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 338–348. DOI: https://doi.org/10.1007/978-1-4612-5695-3_61.

E. W. Dijkstra. 1984. On the nature of computing science. EWD896. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD896.PDF.

E. W. Dijkstra. 1988. On the cruelty of really teaching computing science. EWD1036. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF.

E. W. Dijkstra. 1996. The next fifty years. EWD1243a. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1243a.PDF.

J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. 1979. *LINPACK Users' Guide*. SIAM, Philadelphia. DOI: https://doi.org/10.1137/1.9781611971811.

V. Eijkhout, P. Bientinesi, and R. van de Geijn. 2010. Towards mechanical derivation of Krylov solver libraries. *Procedia Comput. Sci.* 1, 1, 1805–1813. DOI: https://doi.org/10.1016/j.procs.2010.04.202. ISSN 1877-0509. http://www.sciencedirect.com/science/article/pii/S1877050910002036. ICCS 2010.

D. Fabregat-Traver. 2014. *Knowledge-based Automatic Generation of Linear algebra Algorithms and Code*. Ph.D. thesis. RWTH Aachen. http://arxiv.Org/abs/1404.3406.

D. Fabregat-Traver and P. Bientinesi. 2011a. Automatic generation of loop-invariants for matrix operations. In *Computational Science and its Applications, International Conference*. IEEE Computer Society, Los Alamitos, CA, 82–92. DOI: https://doi.org/10.1109/ICCSA.2011.41. http://hpac.cs.umu.se/aices/preprint/documents/AICES-2011-02-01.pdf.

D. Fabregat-Traver and P. Bientinesi. 2011b. Knowledge-based automatic generation of partitioned matrix expressions. In V. Gerdt, W. Koepf, E. Mayr, and E. Vorozhtsov (Eds.), *Computer Algebra in Scientific Computing*, Vol. 6885: Lecture Notes in Computer Science. Springer, Heidelberg, 144–157. DOI: https://doi.org/10.1007/978-3-642-23568-9_12. http://hpac.cs.umu.se/aices/preprint/documents/AICES-2011-01-03.pdf.

D. Gries. 1981. *The Science of Programming*. Springer-Verlag. DOI: https://doi.org/10.1007/978-1-4612-5983-1.

J. A. Gunnels. 2001. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. Ph.D. thesis. Department of Computer Sciences, The University of Texas.

J. A. Gunnels and R. A. van de Geijn. 2001. Formal methods for high-performance linear algebra libraries. In R. F. Boisvert and P. T. P. Tang (Eds.), *The Architecture of Scientific Software: IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, October 2–4, 2000, Ottawa, Canada. Springer, Boston, MA, 193–210. ISBN 978-0-387-35407-1. DOI: https://doi.org/10.1007/978-0-387-35407-1_12.

J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. 2001. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Softw.* 27, 4, 422–455. DOI: http://doi.acm.org/10.1145/504210.504213.

M. Lee and T. M. Low. 2017. A family of provably correct algorithms for exact triangle counting. In *Proceedings of the First International Workshop on Software Correctness for HPC Applications, Correctness'17*. Association for Computing Machinery, New York, NY, 14–20. ISBN 9781450351270. DOI: https://doi.org/10.1145/3145344.3145484.

T. M. Low. 2013. *A Calculus of Loop Invariants for Dense Linear Algebra Optimization*. Ph.D. thesis. Department of Computer Science, The University of Texas at Austin.

T. M. Low, R. A. van de Geijn, and F. G. Van Zee. 2005. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'05*. ACM, New York, NY, 153–163. ISBN 1-59593-080-9. DOI: https://doi.org/10.1145/1065944.1065965.

T. M. Low, V. N. Rao, M. Lee, D. Popovici, F. Franchetti, and S. McMillan. 2017. First look: Linear algebra-based triangle counting without matrix multiplication. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6. DOI: https://doi.org/10.1109/HPEC.2017.8091046.

T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan. 2018. Linear algebraic formulation of edge-centric K-truss algorithms with adjacency matrices. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. DOI: https://doi.org/10.1109/HPEC.2018.8547718.

E. S. Quintana, G. Quintana, X. Sun, and R. van de Geijn. 2001. A note on parallel matrix inversion. *SIAM J. Sci. Comput.* 22, 5, 1762–1771. DOI: https://doi.org/10.1137/S1064827598345679.

R. A. van de Geijn. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press.

F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. 2009. The libflame library for dense matrix computations. *Comp. Sci. Eng.* 11, 6, 56–63. DOI: https://doi.org/10.1109/MCSE.2009.207.

11 Calculational Proofs

Vladimir Lifschitz

11.1 A letter from Dijkstra

The letter from Edsger Dijkstra that I hold in my hands, dated December 11, 1996, begins with the words:

> Dear Vladimir,
> I did my homework, and proved 1.16 through 1.21 straightforwardly.

The "homework" that Edsger referred to was a set of exercises on the use of the axiomatic method handed out to students in my logic class. Each exercise was a property of natural numbers that could be derived from the assumptions about 0 and the successor function $n \mapsto n'$ expressed by Peano axioms, the recursive definition of addition, and the definition of $m \leq n$ as $\exists k(m + k = n)$. In the letter, Edsger showed how such proofs could be presented in the calculational style, described in his book *Predicate Calculus and Program Semantics*, jointly written with Carel Scholten [Dijkstra and Scholten 1990]. That book was the outcome of many years of thinking about

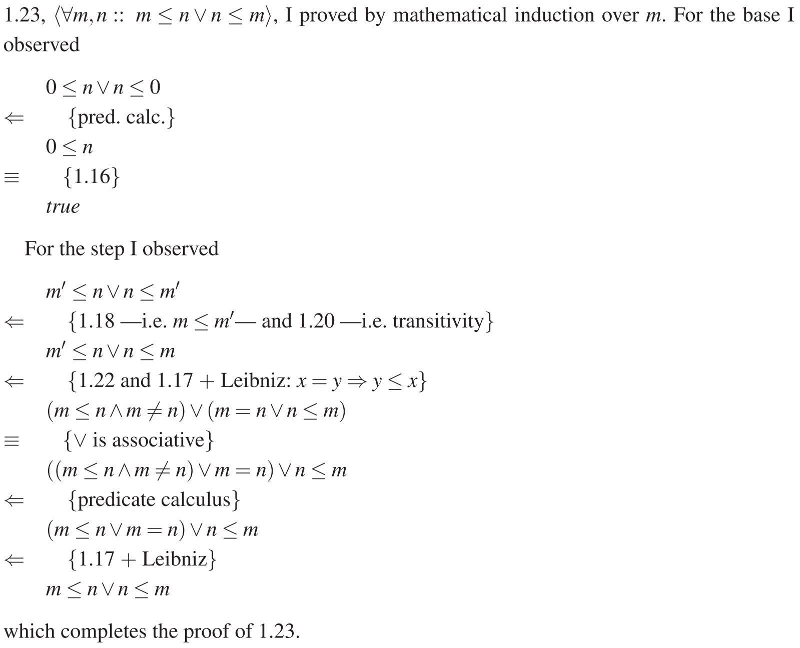
> . . . the streamlining of the mathematical argument, which was inspired by our observation that many—if not most—mathematical arguments we encountered in the literature were unnecessarily complex and lacking in rigour by being incomplete. A few explorations sufficed to convince us that the potential for improvement was dramatic indeed . . . (EWD932a).

Edsger's proof of the formula $m \leq n \vee n \leq m$ shown in Figure 11.1 mentions five other properties of natural numbers:

1.16. $0 \leq n$.

1.17. $n \leq n$.

1.18. $n \leq n'$.



1.23, $\langle \forall m, n ::\ m \leq n \vee n \leq m \rangle$, I proved by mathematical induction over m. For the base I observed

$$\begin{array}{ll}
 & 0 \leq n \vee n \leq 0 \\
\Leftarrow & \quad \{\text{pred. calc.}\} \\
 & 0 \leq n \\
\equiv & \quad \{1.16\} \\
 & \mathit{true}
\end{array}$$

For the step I observed

$$\begin{array}{ll}
 & m' \leq n \vee n \leq m' \\
\Leftarrow & \quad \{1.18 \text{ —i.e. } m \leq m' \text{— and } 1.20 \text{ —i.e. transitivity}\} \\
 & m' \leq n \vee n \leq m \\
\Leftarrow & \quad \{1.22 \text{ and } 1.17 + \text{Leibniz: } x = y \Rightarrow y \leq x\} \\
 & (m \leq n \wedge m \neq n) \vee (m = n \vee n \leq m) \\
\equiv & \quad \{\vee \text{ is associative}\} \\
 & ((m \leq n \wedge m \neq n) \vee m = n) \vee n \leq m \\
\Leftarrow & \quad \{\text{predicate calculus}\} \\
 & (m \leq n \vee m = n) \vee n \leq m \\
\Leftarrow & \quad \{1.17 + \text{Leibniz}\} \\
 & m \leq n \vee n \leq m
\end{array}$$

which completes the proof of 1.23.

Figure 11.1 Dijkstra's calculational proof.

1.20. If $k \leq m$ and $m \leq n$ then $k \leq n$.

1.22. If $m \leq n$ and $m \neq n$ then $m' \leq n$.

It uses induction and consists of two "calculations"—one for the base and the other for the induction step. Each calculation is a list of formulas. Successive members of the list are separated by the symbol ≡ ("is equivalent to") or by ⇐ ("follows from"), along with a "hint" in braces.[1] The hint explains why the relationship expressed by the symbol indeed holds.

The first calculation shows in two steps how the formula to be proved follows from the formula *true*. The second calculation shows how the consequent $m' \leq n \vee n \leq m'$ of the implication to be proved follows from its antecedent $m \leq n \vee n \leq m$. (Many calculational proofs of implications found in the literature are organized in a different way—as chains of formulas leading from the antecedent to the consequent. In such calculations, the symbol ⇐ is replaced by the "implies" symbol ⇒.)

Two hints in the second calculation use Edsger's code word for replacing equals by equals, "Leibniz." These two steps use the implications $m = n \Rightarrow n \leq m$ and $m = n \Rightarrow m \leq n$, which are equivalent to 1.17 "by Leibniz."

[1] According to Dijkstra and Misra [2001], this format is due to W.H.J. Feijen.

After a few examples of this kind, Edsger wrote: "I hope I made my point. I think I would like to recommend that you do some experiments with this calculational proof style since it may be a great improvement over what was available."

Although I have not adopted calculational proof style in my own work, I have spent many hours thinking about it, and this chapter is a summary of my conclusions.

11.2 Dijkstra's Calculational Proofs are Semi-formal

Dijkstra and Scholten characterize their calculational proofs as formal. Reasoning in these proofs is similar to formal reasoning, as this term is understood in work on foundations of mathematics, but there is a difference.

Calculational proofs in *Predicate Calculus and Program Semantics* are similar to formal proofs in two ways. First, intermediate results in such a proof are expressed using propositional connectives and quantifiers, and this use of traditional logical notation is crucial; its function is not merely to abbreviate expressions of a natural language. Think, for instance, of nested occurrences of the equivalence symbol, which are found in the book so many times. Natural languages

> are rather ill-equipped for expressing equivalence. We have, of course, the infamous "if and only if"—where "if" takes care of "follows from" and "only if" of implies—but that is no more than an unfortunate patch: by all linguistic standards, the sentence
>
> > "Tom can see with both eyes if and only if Tom can see with only one eye if and only if Tom is blind."
>
> is—probably for its blatant syntactic ambiguity—total gibberish [Dijkstra and Scholten 1990, chapter 5].

Nested equivalences are hardly ever found in conventional, informal mathematics. Furthermore, calculational proofs, like formal proofs, do not rely on simplifying assumptions ("assume for simplicity," "consider, for instance, the case," "without loss of generality"). They use neither three dots notation $(1, 2, \ldots, 100)$ nor references to pictures.

Second, every step in a calculational proof is a manipulation from some limited repertoire of permissible transitions—as in a formal proof. But in foundational studies, valid justifications for including an assertion in a proof are specified by listing axioms and inference rules, so that the class of formal proofs is defined with mathematical precision, like the class of syntactically correct programs in a programming language. The structure of calculational proofs is described by Dijkstra and Scholten less precisely. Because of this difference, their proofs can be best characterized as semi-formal.

An attempt to identify inference rules that model patterns of calculational reasoning was made by Gries and Schneider [1995], who described a formalization **E** of equational propositional logic and proved its completeness. Axioms and inference rules of **E** are discussed also by Bijlsma and Nederpelt [1998]. A modification of **E** proposed a few years later [Lifschitz 2002] is reviewed in Section 11.5 below; that article shows how adding inference rules for quantifiers can turn **E** into a complete deductive system of first-order logic.

11.3 Calculational Proofs Can Be Smooth

When Dijkstra and Scholten started experimenting with the calculational style, each proof was for them "a well-crafted, but isolated, piece of ingenuity"; later on they developed

> heuristics from which most of the arguments emerged most smoothly (almost according to the principle "There is really only one thing one can do."). The heuristics turned the design of beautiful, formal proofs into an eminently teachable subject [Dijkstra and Scholten 1990, chapter 0].

The idea of smooth argument—reasoning without surprising steps—played an important role in Edsger's thinking. In a note on mathematical methodology (EWD1067), after presenting a solution to a puzzle, Edsger writes: "Some may think this solution cute, surprising, or ingenious, but I would like to point out that there is nothing ingenious about it, because the argument is all but forced." For Edsger in the late eighties, ingenious was bad. In the chapter on the proof format, the authors give examples of calculational proofs containing the implication symbol (⇒), then introduce the consequence symbol (⇐), and explain:

> Before we introduced the consequence, we had many calculations that required considerable clairvoyance to write down in the sense that the motivation for certain manipulations would become apparent several lines further down, where everything would miraculously fall into place. Upon reading them they struck us each time as if a few rabbits had been pulled out of a hat. As soon as we realized that we ourselves had designed those proofs the other way round, we decided to present them in that direction as well, and the symbol ⇐ was introduced. Suddenly, many a manipulation was now strongly suggested by what had already been written down. We were

> almost shocked to see how great a difference such a trivial change in presentation could make and came to fear that the traditional predominance of the implication over the consequence in combination of our habit of reading from left to right has greatly contributed to the general mystification of mathematics [Dijkstra and Scholten 1990, chapter 4].

Proofs of identities that we remember from high school algebra are examples of good proofs from this perspective. We keep simplifying one side of the identity until we arrive at the other side:

$$(a + b)(a - b) = a^2 + ab - ab - b^2 = a^2 - b^2.$$

Symbols do the job; every step is forced; there is nothing surprising or ingenious. Good calculational proofs are similar to this calculation, except that logical formulas take the place of polynomials.

Many mathematicians would be puzzled by the view that smooth reasoning is good mathematics. We prove that there are infinitely many primes by observing that for any list $p_1, \ldots, p_n$ of primes, prime divisors of

$$p_1 \cdot \ldots \cdot p_n + 1 \tag{11.1}$$

do not belong to the list. Expression (11.1) is a rabbit out of a hat, and this rabbit is what makes the proof breathtakingly beautiful. This kind of beauty can be found in Edsger's own work. He surprised us, for instance, by proving a theorem from number theory using a graph with words attached to its vertices (EWD740). That proof is discussed in Chapter 12, written by Jayadev Misra.

My guess is that Edsger's experience with program development has led him to the conclusion that surprising steps are unnecessary and harmful when mathematics is used to develop a program along with the proof of its correctness. That would explain why he considered smooth reasoning so important.

11.4 Calculational Proofs versus Natural Deduction

Natural deduction codifies the use of assumptions in mathematical reasoning. Its invention was motivated by desire

> to construct a formalism that comes as close as possible to actual reasoning. Thus arose a "calculus of natural deduction" [Gentzen 1934].

Derivable objects in a natural deduction system are pairs—a formula F along with a set Γ of assumptions under which F is asserted to hold. Inference rules in such a deductive system are classified into introduction and elimination rules, for instance:

$\wedge$-*introduction*: If F holds under assumptions Γ and G holds under assumptions Δ then conclude that $F \wedge G$ holds under assumptions $\Gamma \cup \Delta$.

$\wedge$-*elimination:* If $F \wedge G$ holds under assumptions Γ then conclude that under assumptions Γ, F holds and G holds.

$\Rightarrow$-*introduction*: If G holds under assumptions Γ then conclude that $F \Rightarrow G$ holds under assumptions $\Gamma \backslash \{F\}$.

$\Rightarrow$-*elimination:* If $F \Rightarrow G$ holds under assumptions Γ and F holds under assumptions Δ then conclude that G holds under assumptions $\Gamma \cup \Delta$.

Constructing a natural deduction proof can be planned according to heuristic rules:

H1. If you want to derive a conjunction $F \wedge G$ then derive F and G, and then use $\wedge$-introduction.

H2. If you assumed or derived $F \wedge G$ then use $\wedge$-elimination to derive F and G.

H3. If you want to derive an implication $F \Rightarrow G$ then assume F, derive G, and then use $\Rightarrow$-introduction.

H4. If you assumed or derived $F \Rightarrow G$ then derive also F, and then use $\Rightarrow$-elimination to derive G.

For example, if we assumed or derived a formula of the form $F \wedge G \Rightarrow H$ then H4 recommends that we try to derive $F \wedge G$ and then use $\Rightarrow$-elimination to derive H. The goal of deriving $F \wedge G$ may be achieved, according to H1, by deriving F and G, and then using $\wedge$-introduction.

To give another example, if our goal is to derive a formula of the form $F \Rightarrow (G \Rightarrow H)$ then H3 recommends that we assume F and try to use this assumption to derive $G \Rightarrow H$. The goal of deriving $G \Rightarrow H$ may be achieved, according to H3, by assuming G and then using this second assumption, along with the assumption F, to derive H. This will be followed by two $\Rightarrow$-introduction steps.

Advice on designing proofs that Dijkstra and Scholten give in their book looks very different—they tell us how to better organize a calculation. For example, they talk about choosing between two possible ways to plan a calculational proof of an implication: is it better to go from the antecedent to the consequent using $\equiv$ and $\Rightarrow$ transitions or to go from the consequent to the antecedent using $\equiv$ and $\Leftarrow$ transitions?

> Experience has taught us that in general the most complicated side is the most profitable one to start with. The probable explanation of this phenomenon is that the opportunities for simplification are usually much more restricted than the possibilities to complicate things: simplification is much more opportunity-driven than "complification" [Dijkstra and Scholten 1990, chapter 4].

In the letter quoted above, Edsger gave another advice: he observed that

$$m = n \Longrightarrow m \leq n$$

is "a more useful statement of the reflexivity of "$\leq$" than $n \leq n$: formulae with more (universally quantified) dummies admit more instantiations and are, hence, a more flexible tool."

In one important case, the calculational approach suggests a strategy different from the natural deduction strategy outlined above. An equivalence $F \equiv G$ can be viewed as shorthand for the conjunction

$$(F \Longrightarrow G) \wedge (G \Longrightarrow F),$$

so that a proof of this equivalence, according to heuristic rules H1 and H3, may be a "ping-pong" argument: assume F and derive G, and then assume G and derive F. The preferred calculational method of proving an equivalence is to construct a chain of equivalent formulas connecting the left-hand side with the right-hand side; there is no need then to prove each of the two implications separately.

Dijkstra and Scholten note that

> [m]any texts on theorem proving seem to suggest that a ping-pong argument is—if not the only way—the preferred way of demonstrating equivalence. This opinion is not confirmed by our experience: avoiding unnecessary ping-pong arguments has turned out to be a powerful way of shortening proofs. At this stage methodological advice is to avoid the ping-pong argument if you can, but to learn to recognize the situations in which it is appropriate [Dijkstra and Scholten 1990, chapter 4].

I found this advice most helpful: my own proofs of if-and-only-if statements are often chains of equivalences, although they do not strictly follow the Feijen—Dijkstra—Scholten format.[2]

It would be a mistake, however, to think that calculational proofs have nothing in common with natural deduction. Heuristic rules H2 and H4 tell us how a

[2] See, for instance, the proof of Lemma 8 in the article by Ferraris et al. [2011].

useful next step in a natural deduction proof is determined by the shape of the formula produced in the previous step, which is a feature of smooth calculational proofs that Edsger valued so much. Furthermore, consider his calculational proof shown in Figure 11.1. It consists of a reference to "mathematical induction over *m*" followed by two calculations. The induction axiom, which is not stated there explicitly, has the form

$$\mathit{Base} \wedge \mathit{Step} \Rightarrow \mathit{Conclusion}.$$

The first calculation proves *Base*; the second proves *Step*. The assertion that providing these calculations completes the proof of 1.23 looks pretty much like the use of $\wedge$-introduction to conclude $\mathit{Base} \wedge \mathit{Step}$, followed by the use of $\Rightarrow$-elimination to arrive at *Conclusion*.

Note also that the proof strategy used in establishing *Base* in this example—building a chain of formulas leading from the goal to the formula *true*—is applicable, in principle, to goal formulas of any syntactic form, including implications. But calculational proofs of implications are usually organized in a different way: they are chains leading from the antecedent to the consequent (or the other way around). This looks pretty much like the use of heuristic rule H3.

11.5 Propositional Calculations

We will show now that some calculations involving propositional formulas correspond to derivations in equational propositional logic.

In this section, *formulas* are propositional formulas formed from atoms and the 0-place connective $\perp$ ("false") using the binary connectives $\equiv$ and $\vee$. Other connectives are treated as abbreviations; for instance, $\neg F$ is shorthand for $F \equiv \perp$.

The axiom schemas of the deductive system **E** express that equivalence is commutative

$$(F \equiv G) \equiv (G \equiv F)$$

and associative

$$(F \equiv (G \equiv H)) \equiv ((F \equiv G) \equiv H);$$

disjunction is commutative

$$F \vee G \equiv G \vee F,$$

associative

$$F \vee (G \vee H) \equiv (F \vee G) \vee H,$$

$$\dfrac{\dfrac{\dfrac{\neg F \equiv G \qquad (F \equiv (\bot \equiv G)) \equiv (\neg F \equiv G)}{F \equiv (\bot \equiv G)} \qquad (\bot \equiv G) \equiv \neg G}{F \equiv \neg G} \qquad (F \equiv \neg G) \equiv \neg(F \equiv G)}{\neg(F \equiv G)}$$

Figure 11.2 A derivation of $\neg(F \equiv G)$ from the assumption $\neg F \equiv G$. Formulas at three leaves are axioms: $(F \equiv (\bot \equiv G)) \equiv (\neg F \equiv G)$ and $(F \equiv \neg G) \equiv \neg(F \equiv G)$ are special cases of the associativity of equivalence; $(\bot \equiv G) \equiv \neg G$ is a special case of the commutativity of equivalence.

and idempotent

$$F \vee F \equiv F;$$

it distributes over equivalence

$$F \vee (G \equiv H) \equiv (F \vee G \equiv F \vee H),$$

and $\bot$ is its neutral element

$$F \vee \bot \equiv F.$$

The system **E** has two inference rules. One of them, when applied to premises F and $G \equiv H$, gives a formula obtained from F by replacing some (or all) occurrences of G by H. The effect of the second rule is opposite—it replaces occurrences of H by G. These rules can be written as

$$\frac{F(G) \quad G \equiv H}{F(H)} \quad \text{and} \quad \frac{F(H) \quad G \equiv H}{F(G)}.$$

(By $F(G)$ we denote the result of substituting G for all occurrences of the placeholder atom X in $F(X)$.) The simpler rules

$$\frac{G \quad G \equiv H}{H} \quad \text{and} \quad \frac{H \quad G \equiv H}{G}$$

are special cases (when $F(X)$ is X).

We will represent derivations in the system **E** as trees, with axioms and assumptions at leaves. For example, Figure 11.2 is a derivation of $\neg(F \equiv G)$ from the assumption $\neg F \equiv G$.

A derivation in the system **E** is *simple* if each of its formulas that does not belong to the leftmost branch is a leaf. Thus, a simple derivation is a single branch with several leaves attached to it. For example, the derivation in Figure 11.2 is simple. Simple derivations have the form

$$\dfrac{\dfrac{\dfrac{\dfrac{F_1 \qquad G_1 \equiv H_1}{F_2} \qquad G_2 \equiv H_2}{\cdots}}{F_{n-1}} \qquad G_{n-1} \equiv H_{n-1}}{F_n} . \tag{11.2}$$

The formula F_{i+1} is obtained from F_i either by replacing occurrences of G_i by H_i or by replacing occurrences of H_i by G_i ($i = 1, \ldots, n-1$).

It is easy to note that simple derivations are disguised calculations in the sense of Dijkstra and Scholten. Derivation (11.2) can be viewed as a chain of transformations

$$\begin{array}{ll}
 & F_1 \\
\equiv & \quad \{G_1 \equiv H_1\} \\
 & F_2 \\
\equiv & \quad \{G_2 \equiv H_2\} \\
\ldots & \\
 & F_{n-1} \\
\equiv & \quad \{G_{n-1} \equiv H_{n-1}\} \\
 & F_n
\end{array}$$

that lead from F_1 to F_n and are justified by the hints $G_i \equiv H_i$. For example, the derivation in Figure 11.2 can be viewed as a calculation establishing the equivalence between $\neg F \equiv G$ and $\neg(F \equiv G)$ using axioms of **E** as hints:

$$\begin{array}{ll}
 & \neg F \equiv G \\
\equiv & \quad \{(F \equiv (\bot \equiv G)) \equiv (\neg F \equiv G)\} \\
 & F \equiv (\bot \equiv G) \\
\equiv & \quad \{(\bot \equiv G) \equiv \neg G\} \\
 & F \equiv \neg G \\
\equiv & \quad \{(F \equiv \neg G) \equiv \neg(F \equiv G)\} \\
 & \neg(F \equiv G).
\end{array}$$

11.6 Reasoning about Predicate Transformers

Dijkstra and Scholten begin their book with a discussion of notational conventions that we have not yet talked about—conventions that

> . . . may strike the reader as a gross overloading of all sorts of familiar operators: for instance, we apply the operators from familiar two-valued logic to operands that in some admissible models may take on uncountably many distinct values [Dijkstra and Scholten 1990, chapter 1].

This aspect of their contribution is essential for the study of predicate transformers, such as the weakest precondition operator.

A *(unary) predicate* on a domain D is a function that maps elements of D to truth values *false*, *true*. We will identify a predicate P with the set of elements of the domain that are mapped by P to *true*. Thus, predicates can be treated as subsets of the domain, and we can write $u \in P$ instead of $P(u)$; this is sometimes convenient. A *predicate transformer* is a function that maps predicates to predicates.[3] An example of a predicate transformer f, where the domain of the (unary) predicates is the set of all integers, is the formula

$$f(P) = \{u : u + 1 \in P\}. \tag{11.3}$$

The domain of integers can be viewed as the state space of programs that involve a single integer variable. We can say that the predicate transformer f defined by (11.3) is the weakest precondition operator for the assignment $u := u + 1$.

The following example demonstrates the usefulness of overloading propositional connectives. A predicate transformer f is *monotonic* if for all predicates P, Q

$$\forall u(P(u) \Rightarrow Q(u)) \Rightarrow \forall u((f(P))(u) \Rightarrow (f(Q))(u)), \tag{11.4}$$

where u is a variable that ranges over elements of the domain. The Dijkstra–Scholten overloading convention allows us to apply the connective $\Rightarrow$ to predicates, rather than truth-valued expressions, and we can write $P \Rightarrow Q$ to represent the predicate $\{u : P(u) \Rightarrow Q(u)\}$. Furthermore, the square brackets around a predicate expression (the "everywhere operator") convert it into the assertion that the predicate maps all elements of the domain to *true*. The antecedent of (11.4) can be

[3] In Chapter 6, written by Reiner Hähnle, given a program, predicate transformers are understood as mappings from formulas to formulas, so that they operate with syntactic objects representing subsets of the domain.

written using this notation as $[P \Rightarrow Q]$. Similarly, the consequent can be written as $[f(P) \Rightarrow f(Q)]$. Thus formula (11.4) turns into the much shorter expression

$$[P \Rightarrow Q] \Rightarrow [f(P) \Rightarrow f(Q)].$$

This is how Dijkstra and Scholten express monotonicity of predicate transformers on page 28 of their book (except that they denote functional application by an infix full stop, so that $f(P)$ becomes $f.P$).

As noted by Rutger Dijkstra [1996] and by Gries and Schneider [1998], the everywhere operator is similar to modal operators, which can be understood as quantifiers binding variables for possible worlds [Garson 2021, Section 6].

To take another example of the use of overloading, a predicate transformer f is *conjunctive* if for all predicates P, Q

$$\forall u((f(R))(u) \equiv (f(P))(u) \land (f(Q))(u)), \tag{11.5}$$

where R stands for $\{u : P(u) \land Q(u)\}$. Overloading $\equiv$ and $\land$ allows us to write (11.5) as

$$[f(R) \equiv f(P) \land f(Q)].$$

The expression that R stands for can be written as $P \land Q$, so that (11.5) turns into

$$[f(P \land Q) \equiv f(P) \land f(Q)].$$

A calculational proof that uses overloading and the everywhere operator is shown in Figure 11.3. There are no quantifiers in this calculation, but its validity can be verified by applying equivalent transformations to formulas with quantifiers. For instance, the last of the four hints, "predicate calculus," indicates the use of the formula

$$[P \Rightarrow Q] \equiv [P \land Q \equiv P],$$

which can be understood as shorthand for the equivalence

$$\forall u(P(u) \Rightarrow Q(u)) \equiv \forall u(P(u) \land Q(u) \equiv P(u)).$$

This equivalence is indeed a theorem of predicate calculus. The authors do not teach us, however, to work with variables ranging over elements of the domain. Instead, they refer the reader to the chapter entitled "The calculus of boolean structures," which includes a long list of useful equivalences that contain the everywhere operator but no explicit quantifiers.

We observe that for any conjunctive predicate transformer f and any predicates P , Q

$$
\begin{array}{ll}
 & [f.P \Rightarrow f.Q] \\
\equiv & \quad \{\text{predicate calculus}\} \\
 & [f.P \wedge f.Q \equiv f.Q] \\
\equiv & \quad \{\, f \text{ is conjunctive}\} \\
 & [f.(P \wedge Q) \equiv f.Q] \\
\Leftarrow & \quad \{\text{Leibniz}\} \\
 & [P \wedge Q \equiv Q] \\
\equiv & \quad \{\text{predicate calculus}\} \\
 & [P \Rightarrow Q]
\end{array}
$$

Figure 11.3 Proof of the theorem: “A conjunctive predicate transformer is monotonic” [Dijkstra and Scholten 1990, chapter 4].

One of the symbols to which Dijkstra and Scholten apply their overloading convention is equality, and this particular deviation from tradition may be confusing. Propositional connectives are traditionally applied to truth-valued expressions, and the reader naturally looks for an explanation whenever they are applied to expressions of any other kind. With the equality symbol, the situation is different: the equality $P = Q$, where P and Q are predicates, has a standard meaning, which is the same as the meaning of $[P = Q]$ in the book. This mismatch may be one of the reasons why many authors found Edsger’s use of brackets hard to understand, as discussed in Chapter 7, written by David Gries.

11.7 Conclusion

The invention of calculational proofs did not revolutionize mathematical writing, as Edsger had perhaps hoped. In fact, a detailed review of *Predicate Calculus and Program Semantics* in *Science of Computer Programming* was highly critical (“the book will not be helpful to those interested in the subject area”) [Börger 1994]. But reading the book has helped many computer scientists understand what a well-written proof is and to learn the art writing good proofs.

In the letter quoted at the beginning of this chapter, Edsger wrote that he saw calculational style

> more and more being adopted: first that was in Ph.D. theses (of candidates I did not know, supervised by colleagues I hardly knew), lately I saw it completely adopted in the textbook “Algebra of Programming” by Bird and de Moor. That book shows another symptom of the acceptance: it just displays its proofs in this format without explaining it and without giving credit for it to the people who designed and promoted it.

Calculational proofs have indeed become standard in work on the formal development of programs [Kaldewaij 1990, Back and von Wright 1998, Backhouse 2003].

A textbook that is widely used in undergraduate teaching [Gries and Schneider 1993] has examples of Dijkstra-style calculations on nearly every page. In the preface, the authors testify that, in their experience, "an equational logic, which is based on equality and Leibniz's rule for substitution of equals for equals" is the logic best suited to be used as a tool in every-day work. I can add that it was a pleasure for me to write proofs in collaboration with students at the University of Texas who were fortunate to have taken Edsger's classes.

Acknowledgements

Many thanks to David Gries, Jayadev Misra, and Fred Schneider for valuable comments on a draft of this chapter.

References

R. Back and J. von Wright. 1998. *Refinement Calculus—A Systematic Introduction*. Graduate Texts in Computer Science. Springer. DOI: https://doi.org/10.1007/978-1-4612-1674-2.

R. Backhouse. 2003. *Program Construction—Calculating Implementations from Specifications*. John Wiley.

L. Bijlsma and R. Nederpelt. 1998. Dijkstra–Scholten predicate calculus: Concepts and misconceptions. *Acta Inf.* 35, 12, 1007–1036. DOI: https://doi.org/10.1007/s002360050150.

E. Börger. 1994. Book review: E.W. Dijkstra, C.S. Scholten, *Predicate Calculus and Program Semantics*. *Sci. Comput. Program.* 23, 91–101. DOI: https://doi.org/10.1016/0167-6423(94)90002-7.

Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag. DOI: https://doi.org/10.1007/978-1-4612-3228-5.

E. W. Dijkstra and J. Misra. 2001. Designing a calculational proof of Cantor's theorem. *Am. Math. Mon.* 108, 5, 440–443. DOI: https://doi.org/10.2307/2695799.

R. M. Dijkstra. 1996. "Everywhere" in predicate algebra and modal logic. *Inf. Process. Lett.* 58, 5, 237–243. DOI: https://doi.org/10.1016/0020-0190(96)00055-5.

P. Ferraris, J. Lee, and V. Lifschitz. 2011. Stable models and circumscription. *Artif. Intell.* 175, 1, 236–263. DOI: https://doi.org/10.1016/j.artint.2010.04.011.

J. Garson. 2021. Modal logic. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy (Summer 2021 Edition)*. Stanford University. https://plato.stanford.edu/entries/logic-modal/.

G. K. E. Gentzen. 1934. Untersuchungen über das logische Schließen. I. *Math. Z.* 39, 176–210. DOI: https://doi.org/10.1007/BF01201353.

D. Gries and F. Schneider. 1993. *A Logical Approach to Discrete Math*. Springer. DOI: https://doi.org/10.1007/978-1-4757-3837-7.

D. Gries and F. Schneider. 1995. Equational propositional logic. *Inf. Process. Lett.* 53, 3, 145–152. DOI: https://doi.org/10.1016/0020-0190(94)00198-8.

D. Gries and F. Schneider. 1998. Adding the everywhere operator to propositional logic. *J. Log. Comput.* 8, 1, 119–129. DOI: https://doi.org/10.1093/logcom/8.1.119.

A. Kaldewaij. 1990. *Programming—The Derivation of Algorithms*. Prentice Hall International Series in Computer Science. Prentice Hall.

V. Lifschitz. 2002. On calculational proofs. *Ann. Pure Appl. Log*. 113, 207–224. DOI: https://doi.org/10.1016/S0168-0072(01)00059-8.

12 An Homage to the Beautiful Mathematical EWDs

Jayadev Misra

Edsger W. Dijkstra was a giant figure in the area of programming theory and practice. He was one of the pioneers of "structured programming" that permeates current programming practices. Later in his career, Dijkstra studied the application of structuring in mathematics. He believed that the ideas of structuring, so successful in programming, would apply in mathematics as well. He experimented with constructing and describing proofs in succinct and elegant styles, which he called "streamlining mathematical arguments."

He had a range of interest in mathematics, from Euclidean geometry to number theory, combinatorics, graph theory, numerical analysis, modern algebra, and logic. In all cases, he wrote down his ideas in the most elegant way he could as a series of notes known as the EWDs.

In this chapter, I discuss a few of his notes. I hope the reader will appreciate his treatments of the chosen problems, the abstractions of the problems, the notations chosen or avoided, the heuristics employed to arrive at a solution, and the orderly presentation of the arguments. In each case, his originality in approaching a problem is a thing of beauty.

It will take a volume to explicate his various contributions in streamlining mathematics. I have chosen to discuss just a few representative EWDs.[1]

[1] The EWD reports [Dijkstra 1980a, 1980b, 1994, 1998a, 1998b, 2002] are reproduced in this chapter with the kind permission of Rutger Dijkstra, who maintains the Dijkstra Archive, https://www.cs.utexas.edu/~EWD/.

12.1 Kruskal's Algorithm for Minimum Spanning Tree

EWD1273 [Dijkstra 1998b] contains a beautiful proof of a classic algorithm in graph theory, Kruskal's [1956] algorithm for constructing a minimum spanning tree. The algorithm scans the edges of a graph in a certain order, accepting or rejecting each edge until the accepted edges form a spanning tree. Dijkstra proposes a lemma, which is trivial to prove, that encompasses both acceptance and rejection criteria; so it leads to a proof of the algorithm in a uniform way.

12.1.1 Minimum Spanning Tree, Kruskal's Algorithm

Minimum spanning tree. A *spanning tree* of a connected undirected graph is a subset of its edges that forms a tree over the nodes of the graph. Given real-valued edge lengths, possibly negative, the length of a spanning tree is the sum of its edge lengths. A spanning tree of minimum length is called a *minimum spanning tree*, henceforth abbreviated as *mst*.

Dijkstra assumes that the edge lengths are distinct. This is an unnecessary assumption, but it makes his proofs easier. One consequence of this assumption is that the mst is unique; in particular, he uses "the mst" instead of "a mst" in his proofs. Dijkstra does not prove uniqueness of the mst; in fact, Kruskal's algorithm is one of the simplest ways to prove it. I show another proof of Kruskal's algorithm in Section 12.1.5 that does not make the assumption of distinct edge lengths.

Kruskal's algorithm. Kruskal's [1956] algorithm starts with an initially empty set of edges M, and adds edges to M one by one. It scans the edges of the graph in the sequence of increasing order of lengths. A scanned edge is added to M if it does not form a cycle with the edges of M, otherwise it is discarded. Scanning of edges continues until M is a spanning tree of the graph. What is not obvious is that M is a minimum spanning tree on termination of the algorithm.

12.1.2 Dijkstra's Lemma

Dijkstra's lemma is best stated in his own words.

> Consider an undirected graph that consists of a single cycle and in which each vertex is marked to be either an A-node or a B-node. Then the number of AB-edges—i.e. edges that connect an A-node with a B-node—is even.

Dijkstra does not prove this lemma, presumably because he felt it to be obvious. An informal proof is as follows. Regard the cycle as a circular path in which each A-node is at street level and B-node one step higher. In traversing the path (in either direction), an edge from a A-node to a B-node requires climbing up a step, from B-node to A-node climbing down a step, and transitioning between two similar nodes requires no climbing, up or down. Since the traversal ends at the same level where it started, the number of upward steps equals the number of downward steps. So, their sum, the number of AB-edges, is even.

A formal proof is equally simple. Assign consecutive indices i, $0 \leq i < n$, to the nodes along the cycle in any direction, and let $i' = i + 1 \bmod n$ be the index of the successor of i. Let $x_i = 0$ if node i is an A-node, $x_i = 1$ for a B-node. I use $\oplus$ for addition modulo 2, which is the same as the exclusive-or operator in propositional logic with 0 being *false* and 1 being *true*. Operator $\oplus$ is commutative and associative, and applied to a set of operands the result is 0 if and only if the number of 1's in the set is even. In particular, $x_i \oplus x_i = 0$, for all i. A formal proof of Dijkstra's lemma is as follows, written in a notation that he and his associates have devised; see Chapter 11 written by Vladimir Lifschitz for a detailed explanation of the notation.

$$\begin{array}{ll}
 & \textit{true} \\
\Rightarrow & \{x_i \oplus x_i = 0, \text{ for each } i\} \\
 & \quad (\oplus i : 0 \leq i < n : x_i \oplus x_i) = 0 \\
\Rightarrow & \{\text{rearrange the terms}\} \\
 & \quad (\oplus i : 0 \leq i < n : x_i \oplus x_{i'}) = 0 \\
\Rightarrow & \{x_i \oplus x_{i'} \text{ is 1 if and only if } (i, i') \text{ is an AB-edge}\} \\
 & \quad \text{the number of AB}-\text{edges is even.}
\end{array}$$

12.1.3 Proof of Kruskal's Algorithm

Dijkstra proves two simple theorems based on his lemma. Both proofs are by contradiction, which he normally never used.

Theorem 12.1 The longest edge of a cyclic path does not belong to the mst.

Proof. Suppose the longest edge e is in the mst. Its removal partitions the mst into two connected components. Label all nodes in one component A, and the other nodes B. Since every node of the graph is in the mst, every node receives a label. The two ends of edge e receive different labels, so it is an AB-edge. From Dijkstra's lemma there is another AB-edge, e', in the cycle that can be used to reconnect the two parts of the mst. Further, the length of e' is smaller than that of e, from the assumption that e is the longest edge and edges have distinct lengths. Therefore, the resulting tree is shorter in length, contradicting that the original tree was the mst. ■

Theorem 12.2 Given an arbitrary labeling of the nodes as A- or B-nodes, the shortest AB-edge belongs to the mst. ■

Proof. Let e be the shortest AB-edge. If the mst does not include e, adding e to the mst creates a cycle. From Dijkstra's lemma, there is another AB-edge, e', in the cycle that is in the mst. Since e is the shortest AB-edge, replacing e' by e in the mst creates a tree of shorter length, a contradiction. ■

We now justify the rejection and acceptance of edges in Kruskal's algorithm based on these theorems.

Rejection: Suppose edge e forms a cycle with the edges in M when it is scanned. All other edges of the cycle are in M, so they are all shorter than e, according to the order of scanning of the edges. So, e is the longest edge in a cycle, and, according to Theorem 12.1, e does not belong in the mst.

Acceptance: The set of edges M partition the nodes into connected components. Suppose edge $e = (x, y)$ does not form a cycle with M when it is scanned. Then x and y belong to different connected components; label one of them A-component by labeling all its nodes as A-nodes and the other a B-component, so that e is an AB-edge. Label other components A or B arbitrarily.

We claim that every edge f scanned prior to e is incident on nodes in a single connected component; so f is not an AB-edge: (1) if f was accepted, it joins two components into a single component, and (2) if f was rejected, it formed a cycle with the edges of M, so its incident nodes are in the same component. Since none of the scanned edges is an AB-edge, e is the first AB-edge to be scanned, so it is the shortest AB-edge. From Theorem 12.2, e belongs to the mst.

12.1.4 Dijkstra–Prim Minimum Spanning Tree Algorithm

Dijkstra published a paper in 1959 [Dijkstra 1959a] in which he described two fundamental graph algorithms, a single-source shortest path algorithm and a spanning tree algorithm; see EWD841a [Dijkstra 1959b] for the background of that paper. The shortest path algorithm is a classic, but the minimum spanning tree algorithm, often referred to as the Dijkstra–Prim algorithm, is not as well-known. I prove the correctness of that algorithm using Dijkstra's lemma.

In the Dijkstra–Prim algorithm there is a single *primary* component (let us call it *prim*; the coincidence in naming is entirely intended) that may consist of one or more nodes. The remaining nodes are each in a different component. Initially, every node is a component by itself, and one of the components is chosen arbitrarily as the primary component. Starting with M as the empty set, in each step add the shortest edge (x, y), where $x \in prim$ and $y \notin prim$, to M and node y to *prim*. Repeat these steps until M is a spanning tree of the graph.

We show that at all points M is a spanning tree of *prim* and a subset of the mst, so on termination it is the mst. The claim holds initially when *prim* has a single node and M is empty. In each subsequent step let every node in *prim* be an A-node and all other nodes B-nodes. From Theorem 12.2 the shortest AB-edge (x, y) belongs to the mst, so after the step M remains a subset of the mst and a spanning tree of *prim*.

12.1.5 A Critique of EWD1273

EWD1273 is beautifully written and argued. Starting with a minor observation about cycles in a graph, "Dijkstra's lemma," Dijkstra manages to prove both rejection and acceptance in Kruskal's algorithm. In the rest of this section, I discuss some of the shortcomings of this note, an alternate proof that is even simpler, and a general theorem that forms the basis of all known minimum spanning tree algorithms.

EWD1273 does not contain any citation, not even for Kruskal's paper [Kruskal 1956]. Dijkstra almost never cited a published paper in an EWD, presumably because he expected the reader to take the trouble of looking up the source material.

EWD1273 does not contain any unnecessary words but omits many necessary ones. He does not describe the necessary properties of a minimum spanning tree that are used in the proof, in particular that (1) adding an edge to a spanning tree creates a cycle, and (2) removing *any* edge from this cycle yields a spanning tree.

Dijkstra assumes that the edge lengths are distinct. This is a strong assumption, which Kruskal did not make. This assumption simplifies Dijkstra's proof because then there is a unique mst. There may be multiple minimum spanning trees when edge lengths are nondistinct. In that case, Theorem 12.1 would read: The longest edge of a cyclic path does not belong to *any* mst, and Theorem 12.2 would be: Given an arbitrary labeling of the nodes as A- or B-nodes, the shortest AB-edge belongs to *some* mst.

An alternate proof. Start with the invariant that M, the set of accepted edges, is a subset of the mst. Initially M is empty, so the invariant is satisfied. Consider a scanned edge e that does not form a cycle with M. Using Theorem 12.2, as Dijkstra does, e belongs to the mst, so adding e to M preserves the invariant. If an edge e forms a cycle with M, it also forms a cycle with the mst because, from the invariant, M is a subset of the mst. So e does not belong to the mst. So, Theorem 12.1, which is used to justify rejection of an edge, can be discarded. ■

It is possible to simplify the proof even further. Dijkstra's lemma asserts the existence of an even number of AB-edges in a cycle. But the subsequent proofs need a much simpler result, that any cycle that contains an AB-edge contains another AB-edge. So, the lemma can be discarded altogether in favor of a direct proof of Theorem 12.2. In fact, most textbooks use a version of Theorem 12.2.

Generalization of Theorem 12.2. Dijkstra's argument covers algorithms that add only one edge at a time to a partially constructed mst. I describe a general theorem, Theorem 12.3 (see Misra [2020]), that permits adding multiple edges to M in each step. Further, it does not assume that the edge lengths are distinct. This generalization forms the basis of all known mst algorithms.

A set of edges M partitions the nodes of a graph into connected components. An edge (x, y), where x and y belong to distinct components X and Y, respectively, is *safe* for X if it is a shortest edge incident on X. Observe that (x, y) that is safe for X may not be safe for Y. Call an edge *safe* if it is safe for *some* component.

Theorem 12.3 Let M be a subset of some mst and E a set of safe edges of distinct lengths. Then $M \cup E$ is a subset of some mst. ■

A generic mst algorithm based on this theorem is as follows. Start with M as the empty set. As long as M is not a spanning tree choose E, as given in the theorem, and add E to M. Different algorithms employ different strategies for choosing E. Kruskal's algorithm scans the edges in increasing order of length and accepts the next edge if and only if it is safe. Dijkstra–Prim algorithm fixes the primary component and chooses a safe edge for this component in each step. These algorithms add a single edge in each step, so E trivially meets the condition of distinct edge lengths. An algorithm from 1926 by Borůvka, perhaps the earliest mst algorithm, which was later modified by Sollin [1965], adds *all* safe edges of distinct lengths in a step.

12.2 Fermat's and Wilson's Theorems

Dijkstra gives two astoundingly beautiful proofs in EWD740 [Dijkstra 1980a] (Fermat's little theorem) and EWD742 [Dijkstra 1980b] (Wilson's theorem). Both are problems in number theory, and he proves them using elementary graph theory. His proofs are by far the simplest and the briefest ones I have seen. Any description of these proofs, including mine, are longer than Dijkstra's original proofs.

12.2.1 Fermat's Little Theorem

The following theorem, known as Fermat's little theorem, is a fundamental result in number theory. The theorem has many applications. Pratt [1975] uses the theorem to certify that a number is prime. It is used in cryptographic protocols, such as the Diffie–Hellman key exchange [Diffie and Hellman 1976].

Theorem 12.4 For any natural number n and prime number p, $n^p - n$ is a multiple of p.

Proof. The following proof is a rewriting of EWD740. For $n = 0$, $n^p - n$ is 0, hence a multiple of p. For positive integer n, take an alphabet of n symbols and construct a directed graph as follows: (1) each node of the graph is identified with a distinct string of p symbols, called a "word," and (2) there is an edge from word x to word y if rotating x by one place to the left yields y. Observe:

(1) No node is on two simple cycles because every node has a single successor and a single predecessor (which could be itself).

(2) Each node is on a cycle of length p because successive p rotations of a word transforms it to itself.

(3) From (2), every simple cycle's length is a divisor of p. Since p is prime, the simple cycles are of length 1 or p.

(4) A cycle of length 1 corresponds to a word of identical symbols. So, exactly n distinct nodes occur in cycles of length 1. The remaining $n^p - n$ nodes occur in simple cycles of length p.

(5) A simple cycle of length p, from the definition of a simple cycle, has p distinct nodes. From (4), $n^p - n$ is a multiple of p. ■

Traditional proofs. There are several ways to prove this theorem, for example, using induction on n. I show a proof purely using number theory. Given natural number n and prime p, it can be shown that for $1 < i < p$, $1 < j < p$ and $i \neq j$, $i{\cdot}n \bmod p \neq j{\cdot}n \bmod p$.

So, $\{i{\cdot}n \bmod p | 1 < i < p\} = \{k | 1 < k < p\}$, that is,

$\Pi(\{i{\cdot}n |\ 1 < i < p\}) \bmod p = \Pi(\{k |\ 1 < k < p\})$, or

$$n^{p-1} \times (p-1)! \overset{\bmod p}{\equiv} (p-1)!.$$

Since prime p does not divide $(p-1)!$, cancel $(p-1)!$ from both sides to get $n^{p-1} \overset{\bmod p}{\equiv} 1$. This implies $n^p \overset{\bmod p}{\equiv} n$, that is, $n^p - n$ is a multiple of p.

A critique of EWD740. The originality and brevity of Dijkstra's solution are remarkable. The criticisms of it are minor. One significant aspect is the use of graph theory in the proof, which was invented almost a century after Fermat stated his result in 1640. In current mathematics, a combinatorial or number-theoretic proof is preferred over one that uses graph theory.

The statement of Fermat's theorem does not appear in Dijkstra's note. The result is stated only at the very end of the note: "i.e. for any n and any prime p, $n^p - n$ is a multiple of p". The "n" in the note is not properly quantified; in particular, Dijkstra assumes that n is a positive integer, from his opening statement that he has an alphabet of n distinct symbols. Fermat's theorem actually covers $n = 0$, a trivial extension, but one worth noting. Another omission occurs in his statement, "Because p successive rotations transform a word into itself, the arrows form cycles of lengths that are divisors of p." (here "arrows" are edges). He should have said *simple cycle*, instead of just *cycle*, because he uses the fact that rotation by a divisor of p also transforms a word to itself.

An alternate proof. A similar alternate proof entirely eliminates the graph. Start with words of length p over an alphabet of size n, as before. Define an equivalence relation over the words: x and y are equivalent if and only if x is a rotation of y. We count the number and size of the equivalence classes.

Define integer q to be a *period* for x if q rotations of x, leftward for positive q and rightward for negative q, yields x. Clearly, 0 is a period for all x, and 1 is a period for x if and only if all symbols in x are identical. Further, given periods q and q' for x and arbitrary integers a and b, $a.q + b.q'$ is a period for x. In particular, a multiple of a period is a period. A period is *simple* if it is not a multiple of another period. For simple period q for x, the words obtained by i rotation of x for each i, $0 \leq i < q$, are all distinct.

The crux of Dijkstra's argument, based on the cycle lengths in a graph, is that for prime p the period of any word x is either 1 or p. The same result can be established using a fundamental theorem of number theory, known as Bézout's identity: given integers m and n, there exist integers a and b such that $a \cdot m + b \cdot n = \text{gcd}(m, n)$, where gcd is the greatest common divisor.

Let q be a simple period for a given x. Setting $m, n := p, q$ in Bézout's identity yields $a \cdot p + b \cdot q = \text{gcd}(p, q)$, for some a and b. Given that both p and q are periods for x, $a \cdot p + b \cdot q$, so $\text{gcd}(p, q)$ is a period for x. Since p is prime, $\text{gcd}(p, q)$ is either 1 or p, and because q is a simple period, $q = 1$ or $q = p$. If $q = 1$, x consists of identical symbols. There are n such words, so $q = p$ for the remaining $n^p - n$ words. Therefore, each of these words belongs to an equivalence class of size p; so, $n^p - n$ is a multiple of p.

12.2.2 Wilson's Theorem

The note EWD742 [Dijkstra 1980b] concerns Wilson's Theorem:

Theorem 12.5 Positive integer p is prime if and only if $(p-1)! \overset{\text{mod}\, p}{\equiv} (-1)$.

Actually, EWD742 never mentions "Wilson's Theorem." It is mentioned in a later note, EWD970 [Dijkstra 1986], that has a sentence that ends, "in the style of my proof of Wilson's Theorem (EWD 742)." So, Dijkstra believed that he had proved Wilson's theorem, though he had proved only one part of the theorem, the "only if" part.

Notation Write $i \oplus j$ for $(i + j \text{ mod } p)$.

Dijkstra's proof. The proof closely follows the structure of his proof for Fermat's theorem. He constructs a graph of $(p-1)!$ nodes in which simple cycles have lengths of either 1 or p, and for prime p exactly $p-1$ nodes are in cycles of length 1. Though the arguments are very similar to those in Fermat's proof, the construction of the graph is more intricate.

A *word* is a permutation of the natural numbers 0 through $p-1$. Two words are equivalent if one can be obtained from the other by applying some number of rotations. Dijkstra calls an equivalence class a *ring*. There are $p!$ words and each word can be rotated in p different ways to obtain members of its ring. So, there are $(p-1)!$ rings.

The *successor* of word $(x_0,\ x_1,\ \cdots\ x_{p-1})$ is $(x_0 \oplus 1,\ x_1 \oplus 1,\ \cdots\ x_{p-1} \oplus 1)$. Observe that successors of the members of a ring form another ring, because if word y results from some number of rotations of x, the successor of y results from the same number of rotations of the successor of x. This allows us to construct a graph in which the nodes correspond to rings and edges correspond to successors. A path of length p starting at node $x = (x_0,\ x_1,\ \cdots\ x_{p-1})$ ends at $(x_0 \oplus p,\ x_1 \oplus p,\ \cdots\ x_{p-1} \oplus p) = x$, that is, the path is a cycle. For prime p, cycles are of length 1 or p.

A word is in a cycle of length 1 if it is a rotation of its successor. Dijkstra gives the proper condition for such words with a terse explanation: "Because a cycle of length 1 corresponds to a ring with a constant difference mod p between each number and its clockwise neighbour and that difference may range from 1 through $p-1$, exactly $p-1$ rings occur in a cycle of length 1." Here "number" refers to a component of a word, and the "clockwise neighbour" of a component is the component to its right in a wraparound fashion, $x_{(i \oplus 1)}$ for x_i. And x is in a cycle of length 1 if there is identical difference mod p between each component and its clockwise neighbor. There are $(p-1)$ possible differences, 1 through $p-1$, so there are $(p-1)$ rings in cycles of length 1. Consequently, there are $(p-1)! - (p-1)$ rings in cycles of length p, and $(p-1)! - (p-1)$ is a multiple of p, or $(p-1)! \overset{\text{mod}\, p}{\equiv} (-1)$.

12.2.3 A Critique of EWD742

As in EWD740 [Dijkstra 1980a], this note is an illustration of Dijkstra's originality, particularly in formulating the problem in terms of permutations over words of appropriate length. My only criticism of the note is that it prefers brevity to clarity. Dijkstra always took great care in introducing mathematical symbols. He often eschewed symbols if he could convey the same ideas in prose. Here he introduces concepts such as ring and successor function without introducing symbols. Judicious use of a few symbols would have helped the explanation in this case.

As I have noted before, Dijkstra only proves that if p is a prime number, $(p-1)! \overset{\mathrm{mod}\, p}{\equiv} (-1)$. The converse can be proved easily without appealing to graph theory, as follows.

For a nonprime p, let $P = \{i|\ 0 \leq i < p\}$. The set of divisors of p, P', is a subset of P, so $p = \Pi\ P'$ divides $\Pi\ P = (p-1)!$. Therefore, $(p-1)! \overset{\mathrm{mod}\, p}{\equiv} 0$, so $(p-1)! \overset{\mathrm{mod}\, p}{\not\equiv} (-1)$.

Reformulating the proof. Dijkstra introduces the notion of a ring, a set of permutations closed under rotation. There is a much simpler abstraction that eliminates words and permutations.

The neighboring symbols within a word remain invariant under rotation; so a word can be specified, up to rotation, by a function f over $P = \{i|0 \leq i < p\}$ where $f(i)$ is the clockwise neighbor of i. Since every element of P is in a word, f is a bijection.

Associate a bijection, corresponding to a ring, with each node of a graph. As Dijkstra has noted, the number of nodes is $(p-1)!$. The successor g of f is defined by $g(i \oplus 1) = f(i) \oplus 1$. The validity of this definition is as follows. Since $f(i)$ is clockwise neighbor of i in a word, $i \oplus 1$ acquires $f(i) \oplus 1$ as its neighbor under g, or $g(i \oplus 1) = f(i) \oplus 1$.

The proof that there are $(p-1)$ bijections that are their own successors is much simpler than Dijkstra's. If f is its own successor, $f(i \oplus 1) = f(i) \oplus 1$. Then f is uniquely determined by its value at any one point, say $f(0)$, which can have any of the $p-1$ possible values in P other than 0. The remaining part of the proof follows Dijkstra's arguments recast for bijections.

12.3 Arithmetic and Geometric Mean

EWD1140 [Dijkstra 1992], EWD1171 [Dijkstra 1994], and EWD1231 [Dijkstra 1996] constitute a series of proofs of the famous *am–gm* inequality that arithmetic mean is at least the geometric mean for a finite nonempty bag of nonnegative real numbers. The problem dates from ancient times, and the proofs are numerous. What is striking is that Dijkstra takes a problem that has been studied for over a millennium and invents a totally original and elegant solution. His solution is among the best that I know. At the end of this section, I give a solution that, I believe, is somewhat simpler.

The problem seems to invite a solution by induction. But the geometric mean of a bag of n items, $n > 0$, has no easy correspondence with that of a bag of $n+1$ items. So, many proofs apply induction on n for bags of sizes 2^n, $n \geq 0$. A well-known proof by Cauchy applies forward–backward induction in which the result is first proved for bags of sizes 2^n for all n, $n > 0$, by induction on n, and then for intermediate bag sizes. A proof by Hardy, Littlewood, and Pólya has a similar structure though it is simpler: they first prove the result for all bags of sizes 2^n then show that if the result holds for all bags of size k, $k > 2$, it holds for all bags of size $k-1$ as well.

At the end of EWD1140 [Dijkstra 1992], Dijkstra says: "The above proof was designed during the last session of the ETAC, and it is recorded because we thought it much nicer than Cauchy's proof that J. Misra subsequently showed us." I fully agree, and also that his proof is superior to that of Hardy, Littlewood, and Pólya.

I analyze EWD1171 [Dijkstra 1994], which is an improvement over EWD1140, and integrate it with the ideas from EWD1231 [Dijkstra 1996]. EWD1231 is based on a suggestion from a student in Dijkstra's undergraduate class that simplifies the proof.

12.3.1 Dikstra's Proof of the am–gm Inequality

For a bag of nonnegative real numbers $\{x_0, x_1, \cdots, x_{n-1}\}$, the arithmetic mean, a, is $(\sum_{i=0}^{n-1} x_i)/n$, and the geometric mean, g, is $(\prod_{i=0}^{n-1} x_i)^{1/n}$. The am–gm inequality says that $a \geq g$. If the bag contains a 0, then $a \geq 0$ and $g = 0$, so $a \geq g$. Henceforth, assume that the bag elements are positive reals, so both a and g are positive. Dijkstra uses symbol c for the geometric mean whereas I use g.

Dijkstra starts EWD1171 by investigating if it is possible to establish a strict inequality, $a > g$, for a bag of all positive reals, and observes that $a = g$ if and only if all elements of the bag are identical to g. This observation is the basis for his proof. Starting with bag B, he transforms it by a series of steps to B'' such that (1) each step preserves the value of g but does not increase a, and (2) all elements of B'' are identical at the end.

Lemma 12.1 Let B be a bag whose arithmetic mean and geometric mean are a and g, respectively, and not all of whose elements are identical. Then there is a bag B' of the same size as B whose arithmetic mean and geometric mean are a' and g', respectively, such that (1) $a' \leq a$, (2) $g' = g$, and (3) B' has more occurrences of g than B.

Proof. Since B has nonidentical elements, neither $(\forall z :: z \leq g)$ nor $(\forall z :: z \geq g)$ holds. That is, there are elements x and y in B such that $x < g < y$. Modify B by replacing the pair (x, y) in B by $(g, x \cdot y/g)$ to obtain B'. The sizes of B and B' are equal.

In EWD1231, Dijkstra uses a very simple argument to prove that $a' \leq a$, which is equivalent to $(g + x \cdot y/g) \leq (x + y)$. He proves the inequalities used in both EWD1140 and EWD1171 using the same method.

$$\begin{array}{ll} & x < g < y \\ \Rightarrow & \{\text{arithmetic}\} \\ & (g - x) \cdot (g - y) < 0 \end{array}$$

$$\begin{array}{ll} \Rightarrow & \{\text{expand and rearrange terms}\} \\ & (g^2 + x \cdot y) < (g \cdot (x + y)) \\ \Rightarrow & \{g > 0.\ \text{Divide both sides by } g\} \\ & (g + x \cdot y/g) \leq (x + y) \end{array}$$

The modification also preserves the geometric mean because $x \cdot y = g \cdot (x \cdot y/g)$. And, it increases the number of g elements by at least 1. ■

Theorem 12.6 For any bag B of positive real numbers its arithmetic mean is at least its geometric mean.

Proof. Let a and g be the arithmetic and geometric mean of B, as before. As long as B has nonidentical elements, apply the step in Lemma 12.1 to construct another bag whose arithmetic mean is less than or equal to a, geometric mean is equal to g, and the number of occurrences of g in it is strictly larger. The given step can be applied only a finite number of times until all elements of the final bag, B'', are identical to g. Writing a'' and g'' for the arithmetic and geometric means of B'', $a'' \leq a$, $g'' = g$, and $a'' = g''$; so $a \geq g$. ■

12.3.2 A Critique of EWD1171

Dijkstra wisely does not attempt to compute the geometric mean of any bag, unlike Cauchy. Geometric mean is far more difficult to manipulate than the arithmetic mean. His transition from EWD1140 [Dijkstra 1992] to EWD1171 [Dijkstra 1994] is an improvement because the former preserves the arithmetic mean whereas the latter has the easier task of preserving the geometric mean.

I wonder why earlier mathematicians did not attempt a proof of this kind. Perhaps repeated transformations in which a particular quantity is never increased (arithmetic mean) while preserving another (the geometric mean) is not a standard heuristic in mathematics, though it is a well-recognized programming concept.

On first reading, I found the proofs, the heuristics, and the program in EWD1171 to be somewhat tangled. But a second reading not only clarified the proof but also the reasons for some of the choices that were made.

A simpler proof using reformulation. The following proof is from Misra [1998], which is a slight simplification of a proof from Beckenbach and Bellman [1961, section 11].

The essential idea is "scaling," multiplying all elements of a bag by a positive real number, something that Dijkstra missed. Scaling by r, $r > 0$, converts the arithmetic mean a and geometric mean g to ra and rg, respectively, and $a \geq g$ iff $ra \geq rg$. Let r be the reciprocal of the product of the elements of the bag, so the product after scaling is 1, and $g = 1$. It is then sufficient to show that the arithmetic mean is at least 1. The equivalent statement of the am–gm inequality is: for any nonempty bag

B of n positive real numbers whose product is 1, the sum of its elements, $sum(B)$, is at least n. The proof is by induction on n.

For $n = 1$, geometric mean is 1 iff $B = \{1\}$, so $sum(B) = 1$. For the inductive step let B have $n+1$ elements, $n \geq 1$. Choose distinct elements x and y from B where x is a smallest and y a largest element. Since the product of the elements of B is 1, $x \leq 1$ and $y \geq 1$. Let B' be the bag obtained by replacing x and y in B by their product, that is, $B' = B - \{x,y\} \cup \{x \cdot y\}$ so that the product in B' is 1. We show that $sum(B) \geq n+1$.

$$
\begin{array}{ll}
 & sum(B) \\
= & \{\text{definition of } B'\} \\
 & sum(B') + x + y - x \cdot y \\
\geq & \{x \leq 1,\ y \geq 1 \Rightarrow (1-x)\cdot(y-1) \geq 0,\ \text{ or } x + y - x \cdot y \geq 1\} \\
 & sum(B') + 1 \\
\geq & \{sum(B') \geq n,\ \text{ from the induction hypothesis}\} \\
 & n + 1
\end{array}
$$

12.4 Hall's Theorem on Distinct Representatives

Dijkstra proves a result in combinatorics in EWD1269 [Dijkstra 1998a] known as Hall's [1935] theorem. There are several proofs of this theorem including one using the theory of flows [Ford and Fulkerson 1962, chapter 2, section 10]. Dijkstra's approach is original. Several years after the circulation of EWD1269, I heard from a graduate student at Stanford who heaped praise on this proof. Apparently, he and his colleagues found this proof superior to the published ones and the ones presented by their professor.

12.4.1 Distinct Representatives

Given a bag B whose members are sets S_i, $0 \leq i < n$, a system of distinct representatives (SDR) is a set of elements t_i, $0 \leq i < n$, where $t_i \in S_i$. B is a *bag* of sets instead of a *set* of sets in order to allow identical sets to be included in B. Hall gives the necessary and sufficient condition for the existence of an SDR for bag B: for every subbag of B, the number of elements in its constituent sets is at least the number of its members, that is, its size.

Dijkstra employs a matrix to represent the given bag: each row represents a member of the bag, a set, and a column an element. A matrix entry of 1 denotes that the element corresponding to the column belongs to the row corresponding to the set, the entry is 0 otherwise. A set of rows *covers* a particular column if and only if at least one row of the set contains a 1 in that column. A matrix is called *happy* if and only if every n of its rows cover at least n distinct columns. The choice of the term “happy” makes me very unhappy indeed. I will follow this terminology though I would have preferred a nonanthropomorphic term.

Dijkstra states Hall’s theorem as follows:

> … in a happy matrix the columns can be ordered in such a way that condition H is met, viz. for each row index i, the i^{th} row has a 1 in the i^{th} column.

Actually, he reorders both rows and columns in his proof.

12.4.2 A Proof of Hall’s Theorem

Hall’s theorem as a matching problem on bipartite graphs. It is easier to state the problem in terms of a bipartite graph, corresponding to the matrix representation of Dijkstra, over nonempty node sets X and Y. Each node in X represents a set and in Y an element. Edge (x, y), where $x \in X$ and $y \in Y$, denotes that $y \in x$. A *matching* is a set of edges that have no common incident nodes. A (X, Y) matching is a matching in which every node of X is incident on some edge in the matching. Thus, a (X, Y) matching denotes an SDR.

For any subset S of X denote its set of neighbors by S', that is,
$S' = \{y | x \in S \text{ and } (x, y) \text{ is an edge}\}$.

Hall condition (HC): Subset S of X *meets* HC if $|S'| \geq |S|$. Graph (X, Y) *meets* HC if every subset of X meets HC.

Theorem 12.7 [Hall] There is a (X, Y) matching if and only if (X, Y) meets HC.

Proof. Dijkstra’s proof is very similar to the proof given here except for terminology. ■

If there is a (X, Y) matching, then every subset S of X has $|S|$ edges in the matching, so S meets HC. The main part of the proof is the converse, that if every subset of X meets HC then there is a (X, Y) matching. If X has just one node, then match it to a neighbor, which exists from HC. I prove the general case by induction on the size of X.

For edge (x, y), $x \in X$ and $y \in Y$, suppose $(X - \{x\}, Y - \{y\})$ meets HC. Then, inductively, there is a $(X - \{x\}, Y - \{y\})$ matching, and adding edge (x, y) to this matching yields a (X, Y) matching.

If $(X - \{x\}, Y - \{y\})$ violates HC, there is a nonempty subset G of $X - \{x\}$ that has fewer than $|G|$ neighbors in $Y - \{y\}$. Since G is a subset of X and (X, Y) meets HC, G has at least $|G|$ neighbors in Y. Therefore, G has exactly $|G|$ neighbors in Y, that is, $|G'| = |G|$; recall that G' is the neighboring set of G.

Given that G is a subset of $X - \{x\}$, $|G| < |X|$. Since (X, Y) meets HC, (G, G') meets HC. Inductively, there is a (G, G') matching. Next, we show that $X - G$ has a matching with its neighbors in Y *without using any element* of G'. These two matchings are disjoint, so together they yield a (X, Y) matching.

Let $H = X - G$, so the set of neighbors of H outside G' is $H' - G'$. Further, G is nonempty, so $|H| < |X|$. We show that any subset S of H has at least $|S|$ neighbors in $H' - G'$, so, inductively, $(H, H' - G')$ has a matching. Let T be the set of neighbors of S in $H' - G'$. Then T and G' are disjoint and $(S \cup G)' = T \cup G'$. Applying HC to $S \cup G$,

$$\begin{array}{ll}
 & |(S \cup G)'| \geq |(S \cup G)| \\
\Rightarrow & \{(S \cup G)' = T \cup G'\} \\
 & |(T \cup G')| \geq |(S \cup G)| \\
\Rightarrow & \{T \text{ and } G' \text{ are disjoint; } S \text{ and } G \text{ are disjoint}\} \\
 & |T| + |G'| \geq |S| + |G| \\
\Rightarrow & \{|G'| = |G|\} \\
 & |T| \geq |S|
\end{array}$$

A critique of EWD1269. Dijkstra uses no mathematical notation in his proof. Instead, he draws figures of matrices after permutations of their rows and columns. This is unusual for him because he felt that a picture is worth a thousand words if only it can replace a thousand words. In fact, I find his proof hard to follow using pictures alone. The proof applies induction on two parts of the problem, similar to the proof given here. Unlike the strategy adopted here, of removing two neighboring nodes x and y, Dijkstra removes only the edge (x, y), which makes his proof harder.

My major criticism is that the use of matrices in the proof is unnecessary, and, in fact, it is a major source of complication. Representation by a bipartite graph avoids any mention of rearrangement because sets are oblivious to permutations. His proof includes a moderate amount of case analysis which could have been completely avoided (I share part of the blame, see the last paragraph of EWD1269 [Dijkstra 1998a]). Finally, no important property of a matrix is used in the proof except that it is an orthogonal arrangement of rows and columns.

Dijkstra says in EWD1269:

> This note presents Hall's Theorem and its proof in an almost visual terminology, whose only defence is that it enabled me to design this proof without pen and paper, while still in bed on an early Sunday morning.

I remonstrated with him at the time that his is not a very clear argument. I have no doubt, though, that he could have constructed a proof as simple as the one given here, or something even simpler, if he had walked from the bed to his desk.

12.4.3 A Simpler Proof of Hall's Theorem

This is a proof by induction on the size of set X. If X is empty, there is a trivial matching. For the general case assume, using induction, that there is a matching over all nodes of X except one node r.

Henceforth $u \overset{n}{-} v$ and $u \overset{m}{-} v$ denote, respectively, that (u, v) is a nonmatching edge and (u, v) a matching edge. An *alternating path* is a simple path of alternating matching and nonmatching edges. Let Z be the subset of nodes of X that are connected to r by an alternating path. Every node of Z except r is connected to a unique node in Z' by a matching edge, from the induction hypothesis; so, there are $|Z| - 1$ nodes in Z' that are so connected. Since Z meets HC, $|Z'| \geq |Z|$. Therefore, there is a node v in Z' that is not connected to any node in Z by a matching edge; let v be a neighbor of u in Z, so $u \overset{n}{-} v$.

Let P be the alternating path connecting r and v,

$$P:\ r = x_0 \overset{n}{-} y_0 \overset{m}{-} x_1 \cdots x_i \overset{n}{-} y_i \overset{m}{-} x_{i+1} \cdots x_t \overset{n}{-} y_t \overset{m}{-} x_{t+1} = u \overset{n}{-} v.$$

If v is incident on a matching edge, say $v \overset{m}{-} w$, then r is connected to w by extending P with the edge $v \overset{m}{-} w$, so $w \in Z$. Then v is connected to w in Z by a matching edge; a contradiction. So, v is not incident on any matching edge.

Flip the edge labels on P from n to m and m to n to obtain a matching in which all nodes of X, including r, are in the matching:

$$r = x_0 \overset{m}{-} y_0 \overset{n}{-} x_1 \cdots x_i \overset{m}{-} y_i \overset{n}{-} x_{i+1} \cdots x_t \overset{m}{-} y_t \overset{n}{-} x_{t+1} = u \overset{m}{-} v.$$

12.5 Coxeter's Rabbit

I discuss EWD1318 [Dijkstra 2002], the last in the EWD series, that contains a beautiful proof of an identity used in plane geometry. Dijkstra wrote this note while he was gravely ill, a few months before his death, yet he retained his usual elegant style.

H.S.M. Coxeter [1961] includes, without proof, the following identity in connection with computing the area of a triangle of sides a, b, and c: Given $s = (a + b + c)/2$,

$$s(s - b)(s - c) + s(s - c)(s - a) + s(s - a)(s - b) - (s - a)(s - b)(s - c) = abc.$$

The obvious approach in a proof is to replace s by $(a + b + c)/2$ in the left-hand side (lhs) of the identity, and then simplify it to abc in the right-hand side (rhs). Dijkstra proposes an entirely novel proof. He shows that the lhs is a "multiple" of c. By symmetry, it is also a "multiple" of a and b; so, the lhs is a "multiple" of abc. He determines the coefficient of multiplicity by computing the value of the lhs for one set of values of a, b, and c, specifically, $a, b, c = 2, 2, 2$, which yields a coefficient of 1. So, the lhs equals abc.

The proof that the lhs is a "multiple" of c is also done without much grind. Consider the first two terms and the last two terms separately, and prove that each is a "multiple" of c. His proof is succinct enough to be savored on its own in EWD1318, which is included in this chapter.

12.5.1 A Critique of EWD1318

Dijkstra's proof, without explanation, treats the lhs and the rhs as *polynomials* in a, b, and c. His use of "multiple" means that one polynomial is divisible by the other. Each of the *factors* in the lhs, s, $(s-a)$, $(s-b)$, $(s-c)$, is a linear combination of variables a, b, and c with nonzero coefficients, and each term in the lhs, such as $s(s-b)(s-c)$ or $(s-a)(s-b)(s-c)$, is a product of exactly *three* factors. Therefore, each term is a polynomial of degree exactly 3, and, consequently, the sum of the terms, the lhs itself, is a polynomial of degree 3. Dijkstra proves that the lhs is a multiple of abc. Division of the lhs by abc, which is itself a polynomial (monomial) of degree 3, therefore, yields a constant coefficient, which he determines to be 1.

The omission of a discussion about the degree of the polynomial is not insignificant. It is easy to misunderstand the argument by interpreting a, b, and c as real numbers. The explanation given above is due to Kees Doets, a retired mathematician from the University of Amsterdam.

The use of polynomials and symmetry in the note are beautiful. I hope the reader appreciates the writing of this beautiful note while Dijkstra was in such poor physical condition.

12.6 Concluding Remarks

The old adage that "beauty is in the eye of the beholder" is only partially true. Studies show that even babies show a preference for beautiful faces (as judged by the adult standard of "beauty"). Most of us would agree that a short proof is preferable to a long one, and a proof that takes less mental effort to master is better than the one that is mentally taxing.

But what makes a proof beautiful? Most people would agree that the following proof of the claim that a 8 × 8 board whose two opposite corner squares have been removed cannot be exactly covered by dominoes is beautiful. Color the squares alternately black and white as in a chess board. Then each domino covers a black and a white square, so any covering by dominoes covers equal number of black and white squares. The opposite corner squares have the same color, so there is no covering for the truncated board. The beauty comes from the unexpected idea of coloring the squares that leads to a concise proof.

The same unexpectedness lends beauty to Dijkstra's proofs of Fermat's and Wilson's theorems, discussed in Section 12.2, converting problems of number theory to graphs over strings. Later in life, Dijkstra was no longer content to construct just a beautiful proof, but to present it in the most accessible manner. The style of rendering proofs as a series of logical statements, interspersed with justifications, dates from the nineties. He once told me during this time that if he wrote his classic shortest path algorithm now it would be a completely different paper. In some of these proofs he does a "pull a rabbit out of a hat," but the magic trick is so amazing that we are not only entertained but try to emulate the magician at home.

We mourn the passing of a great magician.

Acknowledgement

Discussions with Vladimir Lifschitz not only simplified some of the presentations but also improved my understanding of the nature of "beauty" in mathematics.

References

E. Beckenbach and R. Bellman. 1961. *An Introduction to Inequalities*. Mathematical Association of America. DOI: https://doi.org/10.5948/UPO9780883859216.

H. S. M. Coxeter. 1961. *Introduction to Geometry*. John Wiley & Sons, New York.

W. Diffie and M. Hellman. 1976. New directions in cryptography. *IEEE Trans. Inform. Theory* 22, 6, 644–654. DOI: https://doi.org/10.1109/TIT.1976.1055638.

E. Dijkstra. 1959a. A note on two problems in connexion with graphs. *Numer. Math*. 1, 83–89. DOI: https://doi.org/10.1007/BF01386390.

E. W. Dijkstra. 1959b. [Reflections on] A note on two problems in connexion with graphs. EWD841a. https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD841a.PDF.

E. W. Dijkstra. 1980a. A short proof of one of Fermat's theorems. EWD740. https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD740.PDF.

E. W. Dijkstra. 1980b. A sequel to EWD740. EWD742. https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD742.PDF.

E. W. Dijkstra. 1986. F.L. Bauer's conjecture is F.L. Bauer's theorem. EWD970. https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD970.PDF.

E. W. Dijkstra. 1992. The arithmetic mean and the geometric mean. EWD1140. https://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1140.PDF.

E. W. Dijkstra. 1994. The argument about the arithmetic mean and the geometric mean, heuristics included. EWD1171. https://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1171.PDF.

E. W. Dijkstra. 1996. The arithmetic and geometric means once more. EWD1231. https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1231.PDF.

E. W. Dijkstra. 1998a. A simple proof of Hall's theorem. EWD1269. https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1269.PDF.

E. W. Dijkstra. 1998b. On Dijkstra's lemma and Kruskal's algorithm. EWD1273. https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1273.PDF.

E. W. Dijkstra. 2002. Coxeter's rabbit. EWD1318. https://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1318.PDF.

L. Ford, Jr. and D. Fulkerson. 1962. *Flows in Networks*. Princeton University Press.

P. Hall. 1935. On representatives of subsets. *J. London Math. Soc.* 10, 1, 26–30. DOI: https://doi.org/10.1112/jlms/s1-10.37.26.

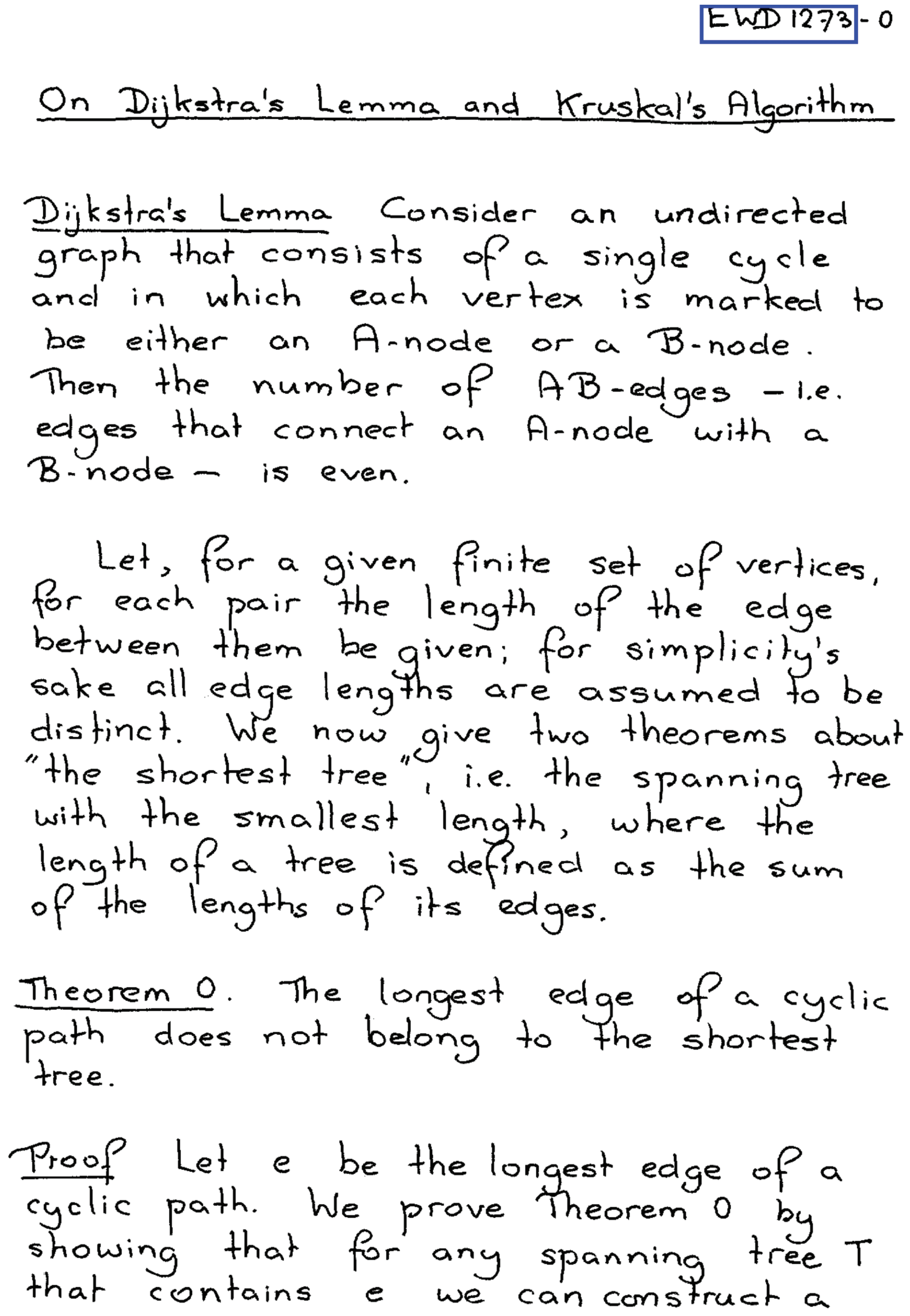
J. B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc*. 7, 1, 48–50. DOI: https://doi.org/10.1090/S0002-9939-1956-0078686-7.

J. Misra. 1998. Arithmetic mean is greater than or equal to geometric mean. https://www.cs.utexas.edu/users/misra/Am-GmInequality.pdf. Unpublished manuscript.

J. Misra. 2020. A fundamental theorem about minimum spanning trees. https://www.cs.utexas.edu/users/misra/MinSpanningTree.pdf. Unpublished manuscript.

V. R. Pratt. 1975. Every prime has a succinct certificate. *SIAM J. Comput.* 4, 3, 214–220. DOI: https://doi.org/10.1137/0204018.

M. Sollin. 1965. La tracé de canalisation. In *Programming, Games, and Transportation Networks*. Wiley.



EWD 1273 - 0

On Dijkstra's Lemma and Kruskal's Algorithm

Dijkstra's Lemma Consider an undirected graph that consists of a single cycle and in which each vertex is marked to be either an A-node or a B-node. Then the number of AB-edges – i.e. edges that connect an A-node with a B-node – is even.

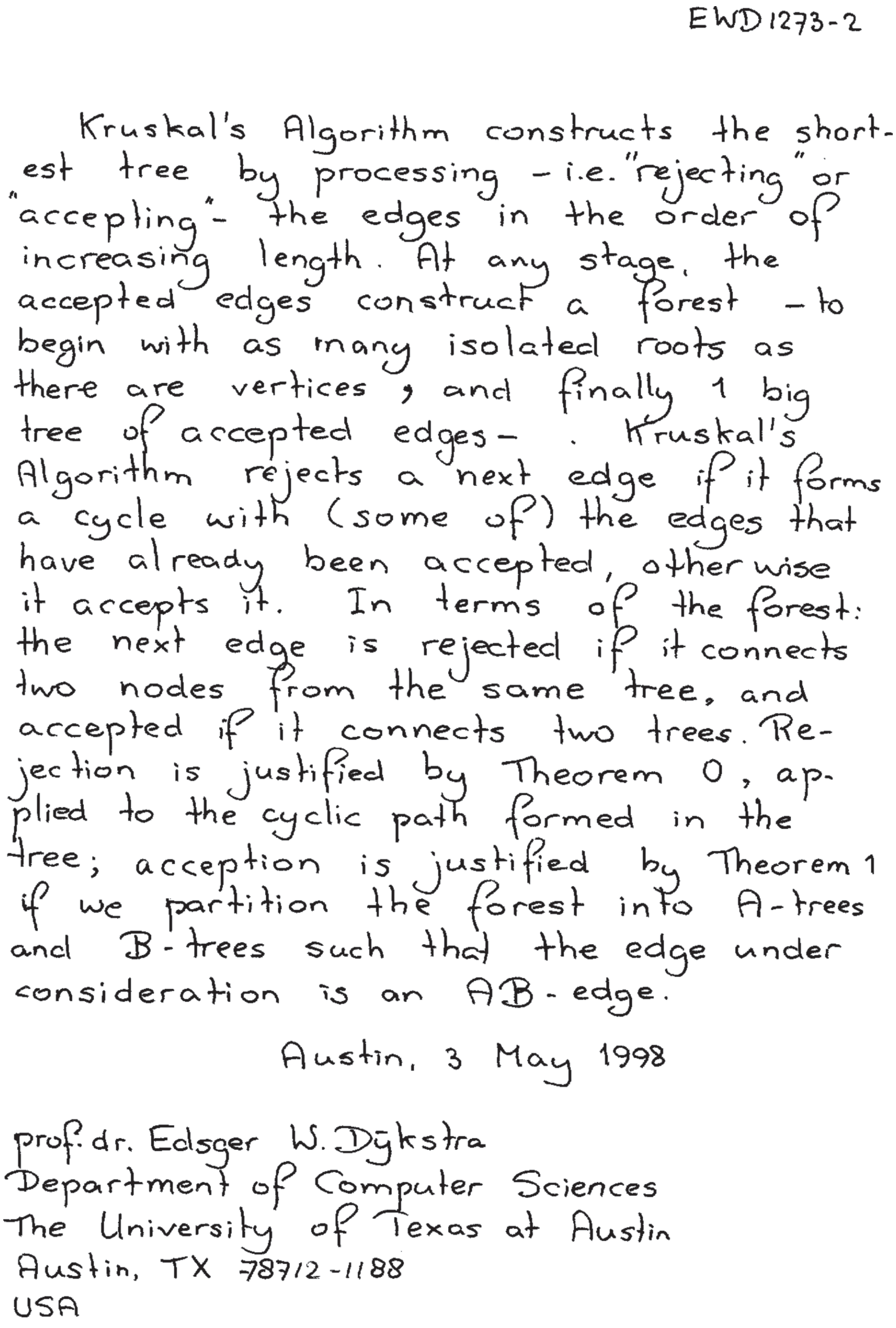
Let, for a given finite set of vertices, for each pair the length of the edge between them be given; for simplicity's sake all edge lengths are assumed to be distinct. We now give two theorems about "the shortest tree", i.e. the spanning tree with the smallest length, where the length of a tree is defined as the sum of the lengths of its edges.

Theorem 0. The longest edge of a cyclic path does not belong to the shortest tree.

Proof Let e be the longest edge of a cyclic path. We prove Theorem 0 by showing that for any spanning tree T that contains e we can construct a shorter spanning tree. Remove edge e from tree T , which thus falls apart into two subtrees which we call A and B respectively ; e being an AB-edge in the cyclic path , Dijkstra's Lemma tells us that the cyclic path contains another AB-edge , which we use to reconnect A and B . The resulting tree is shorter than T because e is the longest edge of the cyclic path. (End of Proof.)

Theorem 1 Partition the vertices into A-nodes and B-nodes. Then the shortest AB-edge belongs to the shortest tree.

Proof Let, for a given partitioning, e be the shortest AB-edge . We prove Theorem 1 by showing that for any spanning tree T that does not contain e , we can construct a shorter spanning tree. Add edge e to tree T , thus creating 1 cyclic path, which contains e ; e being an AB-edge , Dijkstra's Lemma tells us that that cyclic path contains another AB-edge , which we remove. The resulting tree is shorter because e is the shortest AB-edge. (End of Proof.)

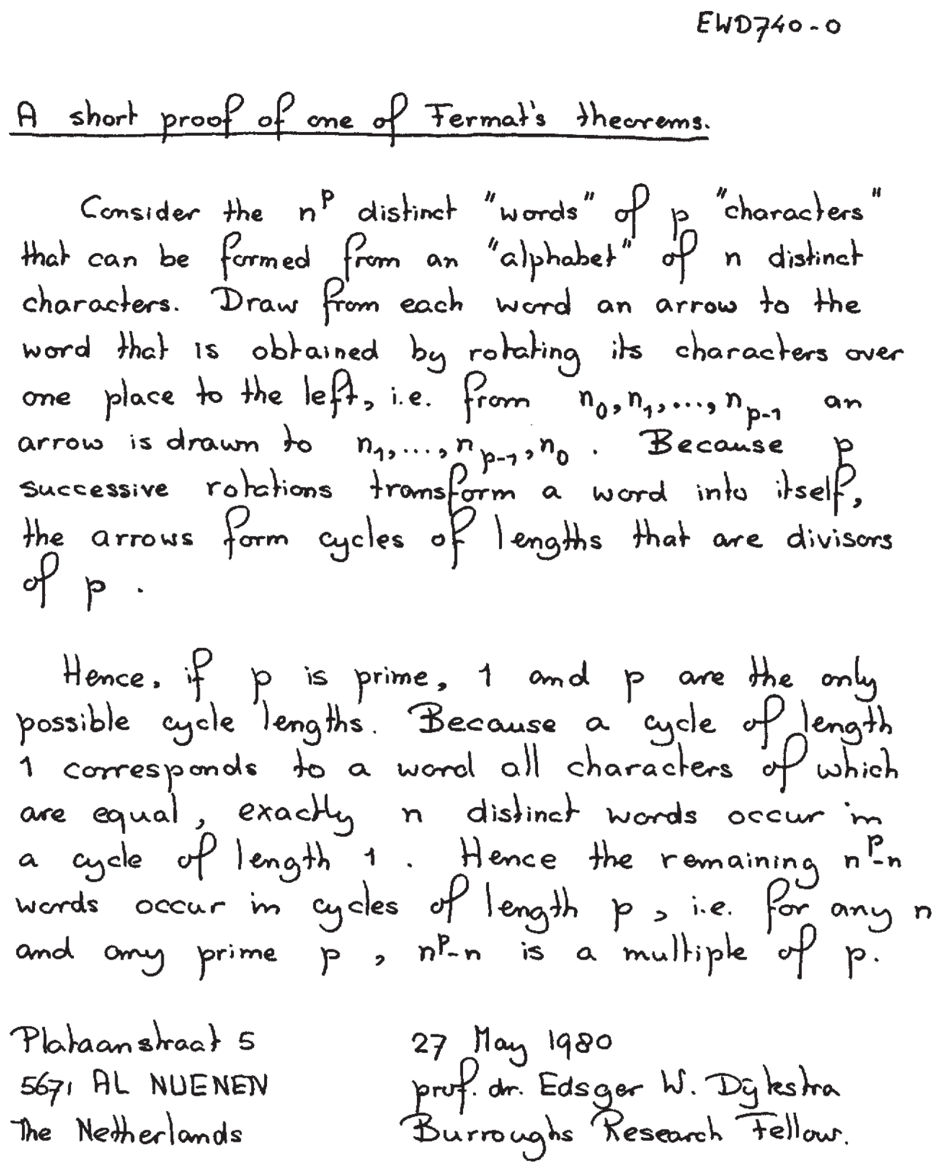


EWD1273-2

Kruskal's Algorithm constructs the shortest tree by processing – i.e. "rejecting" or "accepting" – the edges in the order of increasing length. At any stage, the accepted edges construct a forest – to begin with as many isolated roots as there are vertices, and finally 1 big tree of accepted edges – . Kruskal's Algorithm rejects a next edge if it forms a cycle with (some of) the edges that have already been accepted, otherwise it accepts it. In terms of the forest: the next edge is rejected if it connects two nodes from the same tree, and accepted if it connects two trees. Rejection is justified by Theorem 0, applied to the cyclic path formed in the tree; acception is justified by Theorem 1 if we partition the forest into A-trees and B-trees such that the edge under consideration is an AB-edge.

Austin, 3 May 1998

prof. dr. Edsger W. Dĳkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA



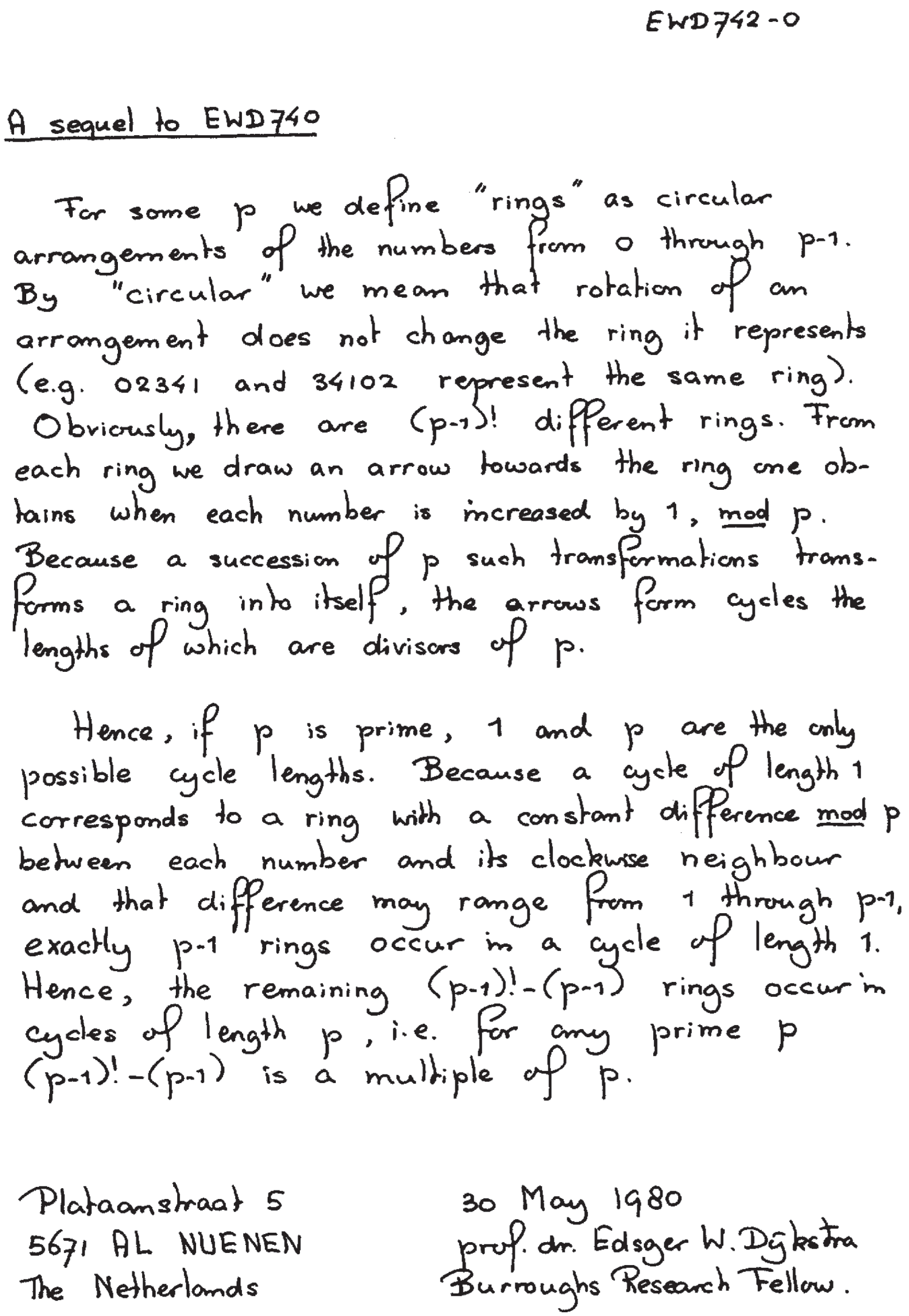
EWD740 - 0

A short proof of one of Fermat's theorems.

Consider the n^p distinct "words" of p "characters" that can be formed from an "alphabet" of n distinct characters. Draw from each word an arrow to the word that is obtained by rotating its characters over one place to the left, i.e. from $n_0, n_1, \dots, n_{p-1}$ an arrow is drawn to $n_1, \dots, n_{p-1}, n_0$. Because p successive rotations transform a word into itself, the arrows form cycles of lengths that are divisors of p .

Hence, if p is prime, 1 and p are the only possible cycle lengths. Because a cycle of length 1 corresponds to a word all characters of which are equal, exactly n distinct words occur in a cycle of length 1 . Hence the remaining $n^p - n$ words occur in cycles of length p , i.e. for any n and any prime p , $n^p - n$ is a multiple of p.

Plataanstraat 5	27 May 1980
5671 AL NUENEN	prof. dr. Edsger W. Dijkstra
The Netherlands	Burroughs Research Fellow.



EWD742 - 0

A sequel to EWD740

For some p we define "rings" as circular arrangements of the numbers from 0 through $p-1$. By "circular" we mean that rotation of an arrangement does not change the ring it represents (e.g. 02341 and 34102 represent the same ring). Obviously, there are $(p-1)!$ different rings. From each ring we draw an arrow towards the ring one obtains when each number is increased by 1, mod p. Because a succession of p such transformations transforms a ring into itself, the arrows form cycles the lengths of which are divisors of p.

Hence, if p is prime, 1 and p are the only possible cycle lengths. Because a cycle of length 1 corresponds to a ring with a constant difference mod p between each number and its clockwise neighbour and that difference may range from 1 through $p-1$, exactly $p-1$ rings occur in a cycle of length 1. Hence, the remaining $(p-1)! - (p-1)$ rings occur in cycles of length p, i.e. for any prime p $(p-1)! - (p-1)$ is a multiple of p.

Plataanstraat 5
5671 AL NUENEN
The Netherlands

30 May 1980
prof. dr. Edsger W. Dijkstra
Burroughs Research Fellow.

EWD1171-0
EWD1171.html

The argument about the arithmetic mean and the geometric mean, heuristics included.

We consider nonempty bags of n non-negative numbers. A bag B's "arithmetic mean" AM.B is defined as one n-th of the sum of the numbers in bag B ; a bag B's "geometric mean" GM.B is defined as the n-th root of the product of the numbers in bag B . An immediate consequence of these definitions is that if we modify the values in a bag without changing their sum (product) , the arithmetic (geometric) mean of the bag remains unchanged. The theorem to be proved is

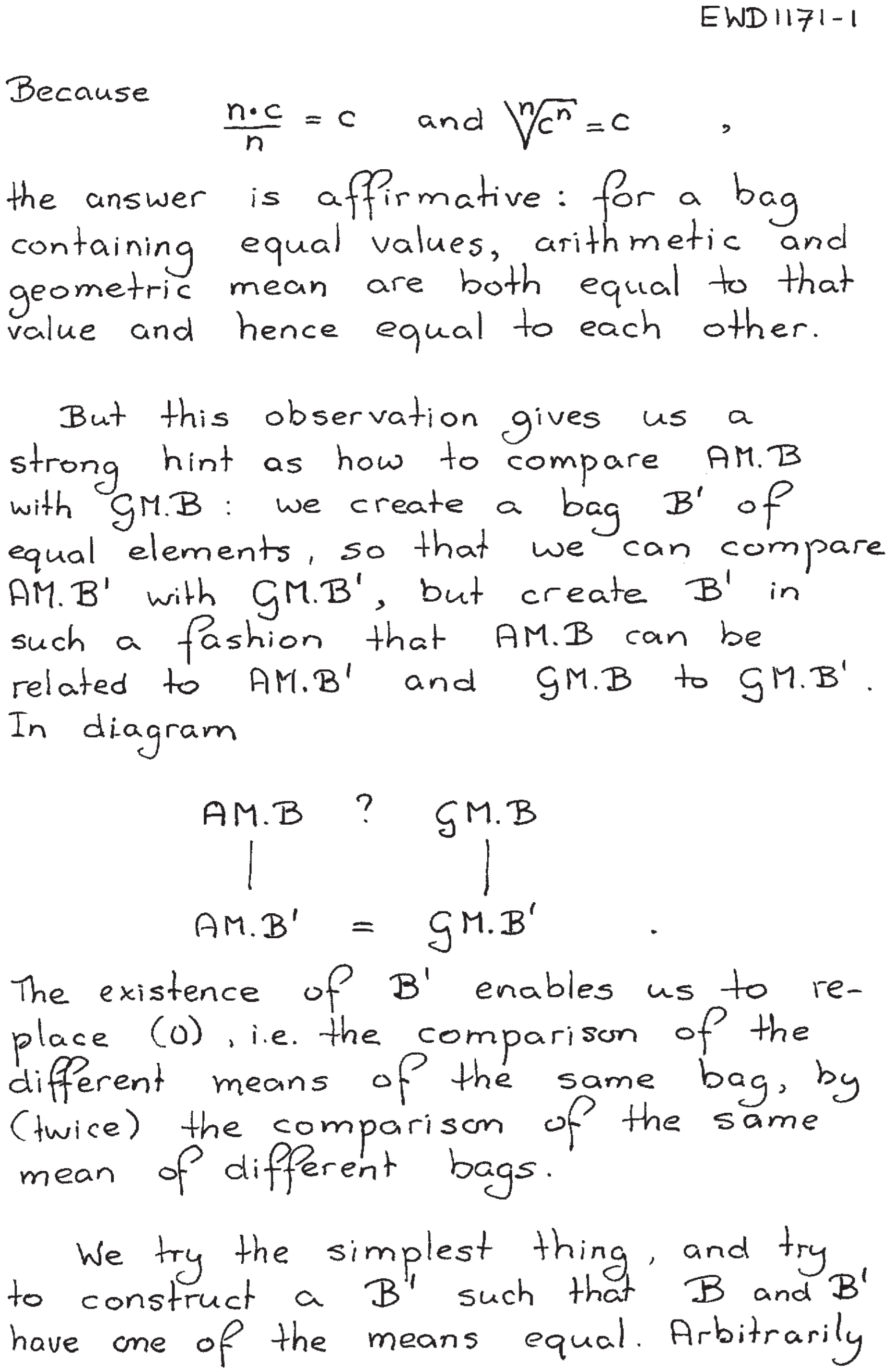
(0) $\quad AM.B \geq GM.B$ for any B .

Note. The way to remember which of the two means is the greater one, is to consider a bag (with $n \geq 2$) that contains exactly one zero. (End of Note.)

* * *

The first thing we ask ourselves is if $AM.B = GM.B$ is possible.

Note If not, the theorem could be strengthened to $AM.B > GM.B$; inspection if a theorem can trivially be strengthened should be a routine matter. (End of Note.)



EWD1171-1

Because

$$\frac{n \cdot c}{n} = c \quad \text{and} \quad \sqrt[n]{c^n} = c \quad ,$$

the answer is affirmative: for a bag containing equal values, arithmetic and geometric mean are both equal to that value and hence equal to each other.

But this observation gives us a strong hint as how to compare AM.B with GM.B : we create a bag B' of equal elements, so that we can compare AM.B' with GM.B', but create B' in such a fashion that AM.B can be related to AM.B' and GM.B to GM.B'. In diagram

$$\begin{tikzcd}
AM.B \arrow[d, dash] & ? & GM.B \arrow[d, dash] \\
AM.B' & = & GM.B'
\end{tikzcd} \quad .$$

The existence of B' enables us to replace (0), i.e. the comparison of the different means of the same bag, by (twice) the comparison of the same mean of different bags.

We try the simplest thing, and try to construct a B' such that B and B' have one of the means equal. Arbitrarily we choose the geometric mean for that purpose.

<u>Note</u> In EWD1140, the arithmetic mean was chosen. (End of Note.)

The overall structure of our proof is now

$$
\begin{array}{ll}
 & AM.B \\
\geq & \{\text{by virtue of the construction of } B'\} \\
 & AM.B' \\
= & \{\text{all elements of } B' \text{ are equal to } c\} \\
 & GM.B' \\
= & \{\text{by virtue of the construction of } B'\} \\
 & GM.B \qquad .
\end{array}
$$

Our remaining task is to construct a program operating on a variable of type bag with B as initial and B' as final value. In order to justify $AM.B \geq AM.B'$, we require

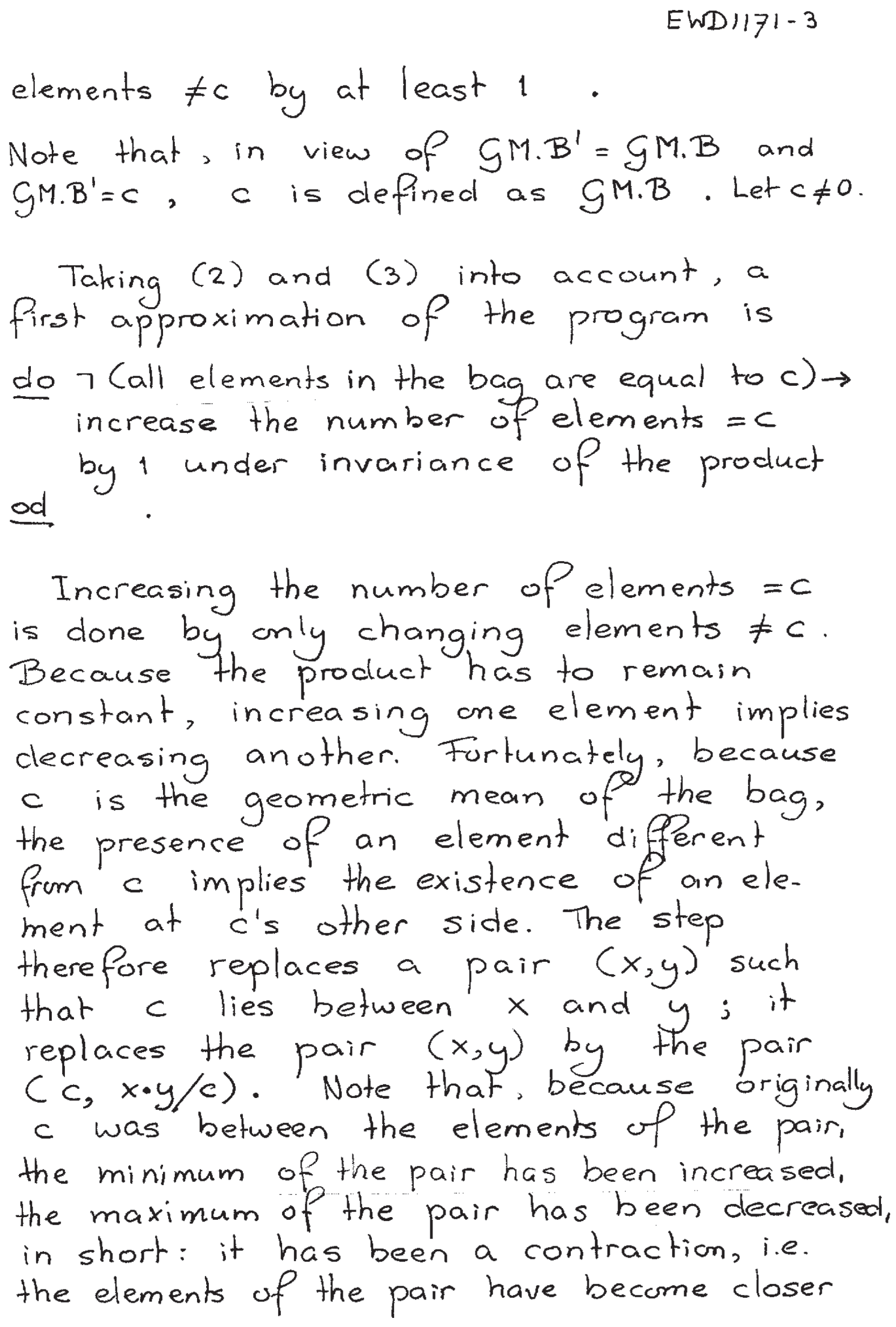
(1) no step increases the sum .

In order to justify $GM.B' = GM.B$, we require

(2) no step changes the product .

In order to assure termination, we require

(3) each step decreases the number of



EWD1171-3

elements $\neq c$ by at least 1 .

Note that, in view of $GM.B' = GM.B$ and $GM.B' = c$, c is defined as $GM.B$. Let $c \neq 0$.

Taking (2) and (3) into account, a first approximation of the program is

```
do ¬(all elements in the bag are equal to c)→
    increase the number of elements =c
    by 1 under invariance of the product
od .
```

Increasing the number of elements $=c$ is done by only changing elements $\neq c$. Because the product has to remain constant, increasing one element implies decreasing another. Fortunately, because c is the geometric mean of the bag, the presence of an element different from c implies the existence of an element at c's other side. The step therefore replaces a pair (x,y) such that c lies between x and y ; it replaces the pair (x,y) by the pair $(c, x{\cdot}y/c)$. Note that, because originally c was between the elements of the pair, the minimum of the pair has been increased, the maximum of the pair has been decreased, in short: it has been a contraction, i.e. the elements of the pair have become closer to each other.

The program now becomes

```
do ¬(all elements in the bag are equal to c) →
   select a pair $(x,y)$ such that $c$ lies
     between them;
   replace this pair by $(c, x \cdot y/c)$
od
```

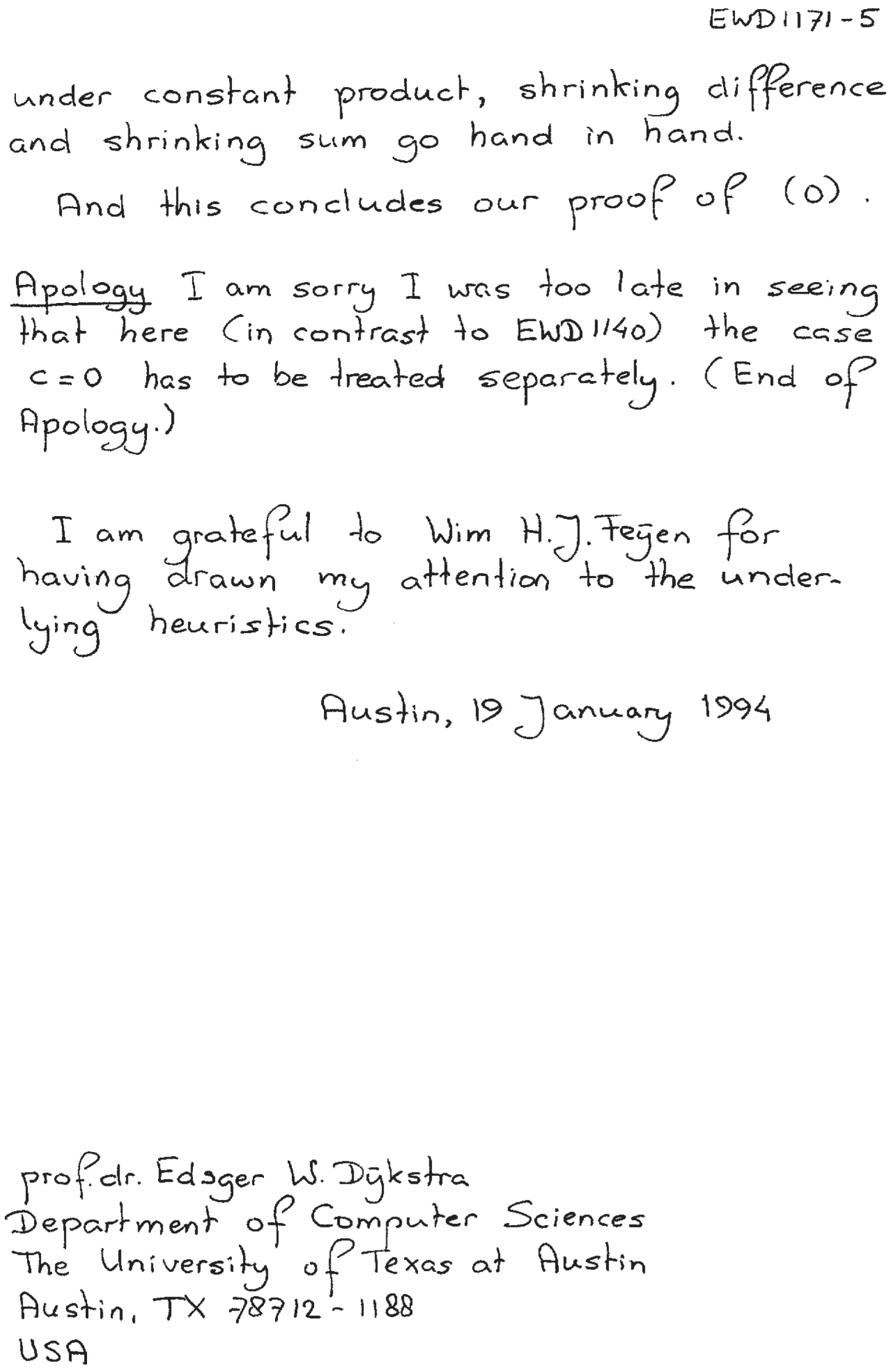
Our remaining proof obligation is (1): we have to show that above step does not increase the sum. (In fact it decreases it.) We have to find a relation between

- the sum, because we have to show that the step decreases it
- the product, because the step leaves it invariant
- the difference, because its decrease distinguishes the step from its inverse.

In view of the product we are talking about a relation of the second degree, in view of the unsigned difference, we have its square, and we come up with

(4) $(x+y)^2 = (x-y)^2 + 4 \cdot x \cdot y$,

which is the rabbit with which EWD1140 started. It expresses that, for positive x, y,



EWD1171-5

under constant product, shrinking difference and shrinking sum go hand in hand.

And this concludes our proof of (0).

<u>Apology</u> I am sorry I was too late in seeing that here (in contrast to EWD1140) the case c = 0 has to be treated separately. (End of Apology.)

I am grateful to Wim H.J. Feijen for having drawn my attention to the underlying heuristics.

Austin, 19 January 1994

prof.dr. Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA

EWD1269 - 0

A simple proof of Hall's Theorem

This note presents Hall's Theorem and its proof in an almost visual terminology, whose only defence is that it enabled me to design this proof without pen and paper, while still in bed on an early Sunday morning.

Take a matrix of 0s and 1s with N rows. We say that a set of rows cover a certain column iff at least one row of the set contains a 1 in that column. A matrix is called happy iff, for any n ($n \leq N$), any n-tuple of its rows covers at least n distinct columns. Hence (i) for $N=0$ the matrix is happy, and a happy matrix (ii) has no row of 0s only, (iii) has at least N columns, and remains happy if (iv) a row is taken away, (v) an all-0 column is taken away, or (vi) a 0 is turned into a 1.

Hall's Theorem states that in a happy matrix the columns can be ordered in such a way that condition H is met, viz. for each row index i, the ith row has a 1 in the ith column.

Remark Above statement of the theorem refers to "the ith row", which is ugly because overspecific: the order of the rows is irrelevant in the sense that subjecting the rows and the equal number of leftmost columns to the same permutation transforms a matrix meeting condition H into a matrix that still satisfies H. Peccavi. (End of Remark.)

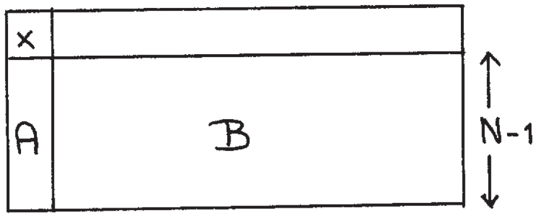
We prove Hall's Theorem (which is an existence theorem) by displaying an algorithm that, given a happy matrix of N rows, rearranges its rows and columns in such a way that condition H is met. Because for N = 0 condition H is met, we can confine our attention to happy matrices with N > 0 .

Since such a matrix contains at least one 1 , we can select a 1 and remove it – i.e. replace it by a (pink) 0 – if the resulting matrix is still happy. We do so repeatedly until –and that moment comes!– we have selected a 1 whose removal would transform a happy matrix into an unhappy one. More specifically this means that the selected element x occurs in some k-tuple of rows such that with x = 1 , the k-tuple covers at least k columns, while with $x=0$, that k-tuple covers less than k columns. Since the transition from $x=1$ to $x=0$ reduces coverages by at most one, we conclude

(0) the number of columns covered by the k-tuple equals k if $x=1$, and equals $k-1$ if $x=0$.

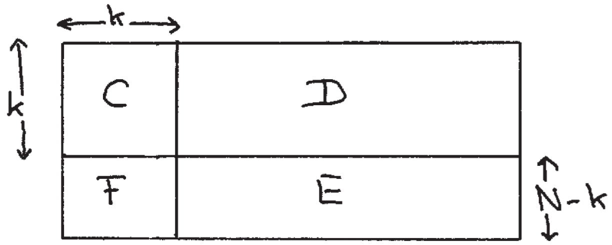
We now distinguish two cases.

<u>$k=N$</u> By permuting rows and columns, we move x to the top-left corner:



Because removal of the 1 at x would reduce the coverage of the whole matrix, the truncated column A consists of 0s only. Therefore, the set of columns covered by an n-tuple of rows from B remains the same when the rows are extended with their element from A; consequently the happiness of the whole matrix implies that B is happy. The algorithm is applied recursively to matrix B , which has only N-1 rows.

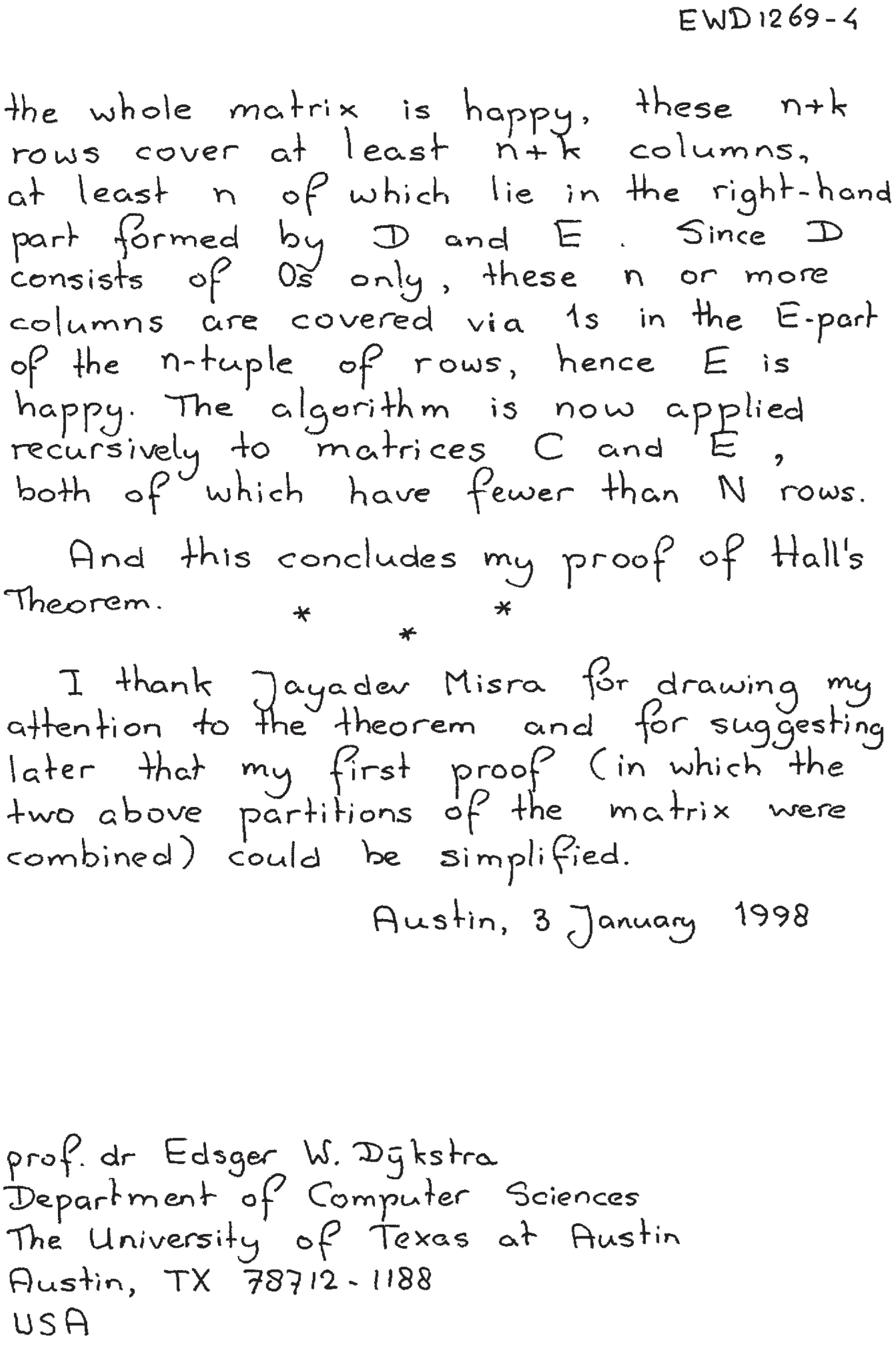
$k < N$ By permuting rows and colums, we move the rows of k-tuple to the top and the k columns they cover to the left:



Element x occurs in C . Since (0) states that the k top rows cover the k left columns and nothing more, submatrix D consists of 0s only. We can now conclude the happiness of C and E as follows.

Because the whole matrix is happy, so is the top part formed by C and D, but since D , consisting of 0s only, does not contribute to the coverage, C is happy all by itself.

To establish the happiness of E , we consider an n-tuple of rows from the lower part formed by F and E , and extend this n-tuple with the k rows of the top part. Because



EWD1269-4

the whole matrix is happy, these n+k rows cover at least n+k columns, at least n of which lie in the right-hand part formed by D and E . Since D consists of 0s only, these n or more columns are covered via 1s in the E-part of the n-tuple of rows, hence E is happy. The algorithm is now applied recursively to matrices C and E , both of which have fewer than N rows.

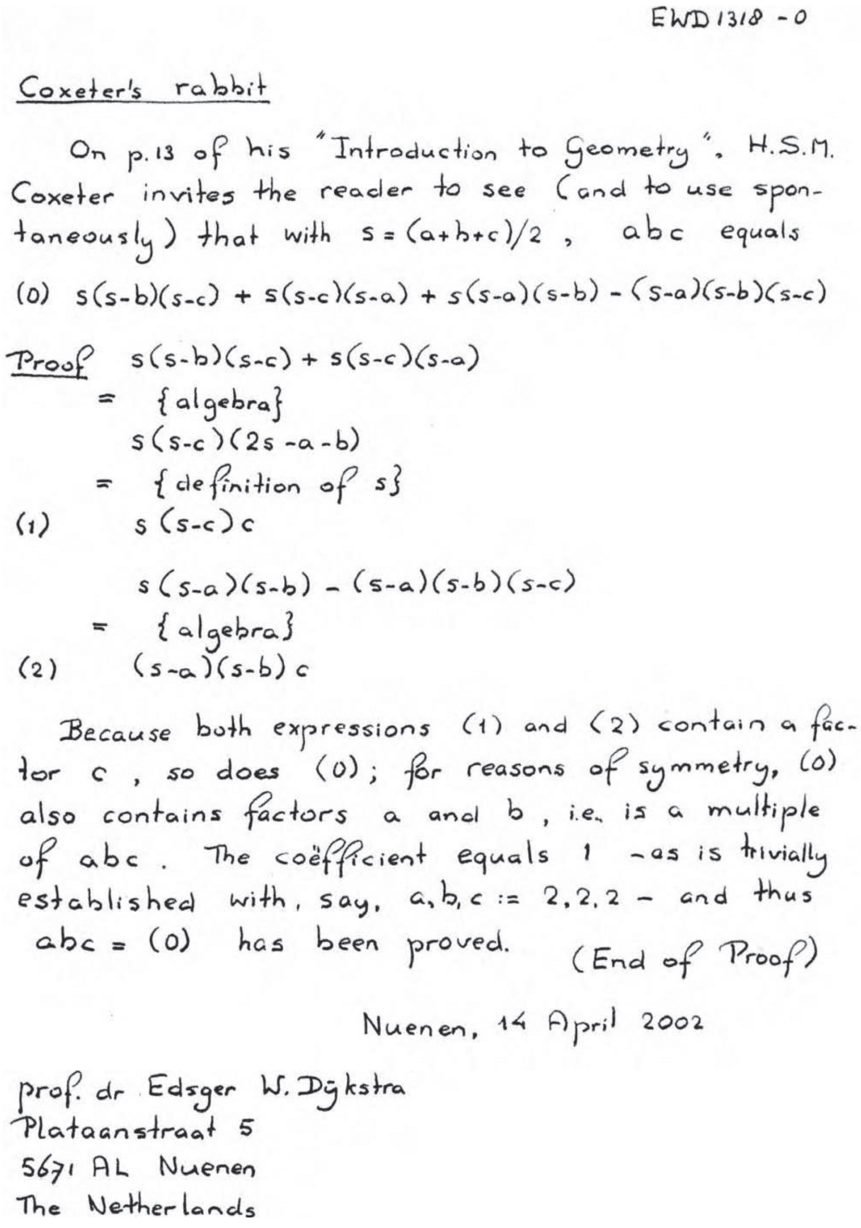
And this concludes my proof of Hall's Theorem.

* * *

I thank Jayadev Misra for drawing my attention to the theorem and for suggesting later that my first proof (in which the two above partitions of the matrix were combined) could be simplified.

Austin, 3 January 1998

prof. dr Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA



EWD 1318 - 0

Coxeter's rabbit

On p.13 of his "Introduction to Geometry". H.S.M. Coxeter invites the reader to see (and to use spontaneously) that with $s = (a+b+c)/2$, abc equals

(0) $s(s-b)(s-c) + s(s-c)(s-a) + s(s-a)(s-b) - (s-a)(s-b)(s-c)$

Proof $s(s-b)(s-c) + s(s-c)(s-a)$

$=$ {algebra}

$s(s-c)(2s-a-b)$

$=$ {definition of s}

(1) $s(s-c)c$

$s(s-a)(s-b) - (s-a)(s-b)(s-c)$

$=$ {algebra}

(2) $(s-a)(s-b)c$

Because both expressions (1) and (2) contain a factor c, so does (0); for reasons of symmetry, (0) also contains factors a and b, i.e. is a multiple of abc. The coëfficient equals 1 – as is trivially established with, say, $a,b,c := 2,2,2$ – and thus $abc = (0)$ has been proved. (End of Proof)

Nuenen, 14 April 2002

prof. dr Edsger W. Dijkstra
Plataanstraat 5
5671 AL Nuenen
The Netherlands

PART III

SELECTED PAPERS

Edsger W. Dijkstra, June 1982, Nuenen, The Netherlands

13

A Note on Two Problems in Connexion with Graphs

E. W. Dijkstra

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Problem 1 Construct the tree of minimum total length between the n nodes. (A tree is a graph with one and only one path between every two nodes.)

In the course of the construction that we present here, the branches are subdivided into three sets:

I. the branches definitely assigned to the tree under construction (they will form a subtree);

II. the branches from which the next branch to be added to set I, will be selected;

III. the remaining branches (rejected or not yet considered).

The nodes are subdivided into two sets:

A. the nodes connected by the branches of set I,

B. the remaining nodes (one and only one branch of set II will lead to each of these nodes).

We start the construction by choosing an arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. To start with, set I is empty. From then onwards we perform the following two steps repeatedly.

Step 1. The shortest branch of set II is removed from this set and added to set I. As a result one node is transferred from set B to set A.

Originally published in Numerische Mathematik 1, 269–271 (1959).

Step 2. Consider the branches leading from the node, that has just been transferred to set A, to the nodes that are still in set B. If the branch under consideration is longer than the corresponding branch in set II, it is rejected; if it is shorter, it replaces the corresponding branch in set II, and the latter is rejected.

We then return to step 1 and repeat the process until sets II and B are empty. The branches in set I form the tree required.

The solution given here is to be preferred to the solution given by J. B. Kruskal [1] and those given by H. Loberman and A. Weinberger [2]. In their solutions all the — possibly $\frac{1}{2}n(n-1)$ — branches are first of all sorted according to length. Even if the length of the branches is a computable function of the node coordinates, their methods demand that data for all branches are stored simultaneously. Our method only requires the simultaneous storing of the data for at most n branches, viz. the branches in sets I and II and the branch under consideration in step 2.

Problem 2 Find the path of minimum total length between two given nodes P and Q.

We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R. In the solution presented, the minimal paths from P to the other nodes are constructed in order of increasing length until Q is reached.

In the course of the solution the nodes are subdivided into three sets:

A. the nodes for which the path of minimum length from P is known; nodes will be added to this set in order of increasing minimum path length from node P;

B. the nodes from which the next node to be added to set A will be selected; this set comprises all those nodes that are connected to at least one node of set A but do not yet belong to A themselves;

C. the remaining nodes.

The branches are also subdivided into three sets:

I. the branches occurring in the minimal paths from node P to the nodes in set A;

II. the branches from which the next branch to be placed in set I will be selected; one and only one branch of this set will lead to each node in set B;

III. the remaining branches (rejected or not yet considered).

To start with, all nodes are in set C and all branches are in set III. We now transfer node P to set A and from then onwards repeatedly perform the following steps.

Step 1. Consider all branches r connecting the node just transferred to set A with nodes R in sets B or C. If node R belongs to set B, we investigate whether the use of branch r gives rise to a shorter path from P to R than the known path that uses the corresponding branch in set II. If this is not so, branch r is rejected; if, however, use of branch r results in a shorter connexion between P and R than hitherto obtained, it replaces the corresponding branch in set II and the latter is rejected. If the node R belongs to set C, it is added to set B and branch r is added to set II.

Step 2. Every node in set B can be connected to node P in only one way if we restrict ourselves to branches from set I and one from set II. In this sense each node in set B has a distance from node P: the node with minimum distance from P is transferred from set B to set A, and the corresponding branch is transferred from set II to set I. We then return to step 1 and repeat the process until node Q is transferred to set A. Then the solution has been found.

Remark 1 The above process can also be applied in the case where the length of a branch depends on the direction in which it is traversed.

Remark 2 For each branch in sets I and II it is advisable to record its two nodes (in order of increasing distance from P), and the distance between P and that node of the branch that is furthest from P. For the branches of set I this is the actual minimum distance, for the branches of set II it is only the minimum thus far obtained.

The solution given above is to be preferred to the solution by L. R. Ford [3] as described by C. Berge [4], for, irrespective of the number of branches, we need not store the data for all branches simultaneously but only those for the branches in sets I and II, and this number is always less than n. Furthermore, the amount of work to be done seems to be considerably less.

References

[1] Kruskal jr., J. B.: On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem. Proc. Amer. Math. Soc. 7, 48–50 (1956).

[2] Loberman, H., and A. Weinberger: Formal Procedures for Connecting Terminals with a Minimum Total Wire Length. J. Ass. Comp. Mach. **4**, 428–437 (1957).

[3] Ford, L. R.: Network flow theory. Rand Corp. Paper, P-923, 1956.

[4] Berge, C.: Théorie des graphes et ses applications, pp. 68–69. Paris: Dunod 1958.

Mathematisch Centrum
2e Boerhaavestraat 49
Amsterdam-O

(*Received June 11, 1959*)

14 Recursive Programming

E. W. Dijkstra

14.1 The Aim

If every subroutine has its own private fixed working spaces, this has two consequences. In the first place the storage allocations for all the subroutines together will, in general, occupy much more memory space than they ever need *simultaneously*, and the available memory space is therefore used rather uneconomically. Furthermore—and this is a more serious objection—it is then impossible to call in a subroutine while one or more previous activations of the same subroutine have not yet come to an end, without losing the possibility of finishing them off properly later on.

We intend to describe the principles of a program structure for which these two objections no longer hold. In the first place we sought a means of removing the second restriction, for this essentially restricts the admissable structure of the program; hence the name “Recursive Programming”. More efficient use of the memory as regards the internal working spaces of subroutines is a secondary consequence not without significance. The solution can be applied under perfectly general conditions, e.g. in the structure of an object program to be delivered by an ALGOL 60 compiler. The fact that the proposed methods tend to be rather time consuming on an average present day computer, may give a hint in which direction future design might go.

14.2 The Stack

The basic concept of the method is the so-called *stack.* One uses a stack for storing a sequence of information units that increases and decreases at one end only, i.e. when a unit of information that is no longer of interest is removed from the

Report MR 33 of the Computation Department of the Mathematical Centre, Amsterdam. Originally published in Numerische Mathematik 2, 312–318 (1960).

stack, then this is always the most recently added unit still present in the stack. For example, one can construct a stack as follows: a number of successive storage locations are set aside for the stack and also an administrative quantity, the "stack pointer", that always points to the first free place in the stack (i.e. the value of the stack pointer may be defined as the address of the first free location in the stack; if the stack is empty to start off with, the stack pointer is equal to the start address of the portion of the memory reserved for the stack). When an information unit is added to the stack the stack pointer indicates where this unit must be stored and when this has been done, the value of the stack pointer is accordingly increased. To remove one or more information units from the stack the value of the stack pointer is suitably decreased.

If we mark off, on a time axis, the moments when a unit is added to or removed from the stack, by using an opening bracket for the addition of a unit and a closing bracket for its removal, then we obtain a correctly nested bracket structure, in which opening and closing brackets form pairs in the same way as they do in a normal algebraic expression involving brackets. This is closely related to the circumstance that we can use a stack for storing the intermediate results formed in the evaluation of an arbitrary algebraic expression by means of elementary algebraic operations. In this case our interest is always restricted to the most recent element in the stack. As the intermediate results are used only once, use of an element implies its removal from the stack.

A simple example may be given in illustration; the successive stack locations are indicated by $v_0, v_1, v_2, \ldots$ etc. The evaluation of

$$A + (B - C) \times (D/E + F)$$

can be split up into: $v_0 := A; v_1 := B; v_2 := C; v_1 := v_1 - v_2; v_2 := D; v_3 := E; v_2 := v_2/v_3; v_3 := F; v_2 := v_2 + v_3; v_1 := v_1 \times v_2; v_0 := v_0 + v_1$; the required result is formed in v_0. The reader will be aware that the operations used are of only two different types. If we denote the value of the stack pointer by k, they are:

1. selecting a new (explicitly mentioned) number, say X, which process is described by $v_k := X; k := k + 1$; and
2. performing an arithmetic operation, say OP, which consists of

$$k := k - 1; \qquad v_{k-1} := v_{k-1} OP\, v_k.$$

If we refer to type 1 by the name of the selected variable and to type 2 by the operator in question, then we can also record the program by means of the symbol sequence:

$$A,\ B,\ C,\ -,\ D,\ E,\ /,\ F,\ +,\ \times,\ +.$$

In this description the v's and, what is more important, specific values of the stack pointer no longer appear. Here it is unnecessary to specify the values of the stack pointer in the text, as the computer can keep track of its value during execution of the program: the v's have become completely *anonymous* again, just as anonymous as they originally were.

The above is well known (see for instance [1]) and so elegant that we could not refrain from trying to extend this technique by consistent application of its principles[1]. Let us consider for a moment the operation for the selection of an argument, e.g. the third "$v_2 := C$". This operation can be executed without further claim for memory space, as we assume that the numerical value of C can already be found in the memory. If, instead of C, a compound term had occured in the expression, e.g. $C = (P/(Q - R + S \times T))$, then we would have used v_2 up to v_5 for the calculation of this subexpression, but the nett result of this piece of program would still be $v_2 := P/(Q - R + S \times T)$ or $v_2 := C$. In other words, it is immaterial to the "surroundings" in which the value C is used, whether the value C can be found ready-made in the memory, or whether it is necessary to make temporary use of a number of the next stack locations for its evaluation. When a function occurs instead of C and this function is to be evaluated by means of a subroutine, the above provides a strong argument for arranging the subroutine in such a way that it operates in the first free places of the stack, in just the same way as a compound term written out in full.

14.3 Stacked Reservations

In the evaluation of an algebraic expression as described above, we stack *numerical information:* the next place in the stack only becomes occupied when we actually fill in an intermediate result, it becomes vacant as soon as its contents have been used. We now consider the case that this expression—which occurs in what we shall refer

[1] Without doubt, in view of the vivid interest in the construction of compilers, at least some of these extensions have been envisaged by others but the author was unable to trace any publication that went further than [1]. The referee was so kind to send him a copy of the report „Gebrauchsanleitung für die ERMETH“ of the Institut für Angewandte Mathematik der ETH, Zürich, a description by Heinz Waldburger of a specific program organisation in which similar techniques as developed by Professor H. Rutishauser in his lectures are actually incorporated. The author of the present paper thinks, however, that there the principle of recursiveness has not been carried through to this ultimate consequences which leads to logically unnecessary restrictions like the impossibility of nesting intermediate returns and the limitation of the order of the subroutine jump (cf. section F 44 of the report).

to as the (relative) main program—contains a function that is to be evaluated by means of a subroutine. As far as the main program is concerned, *all* variables that the subroutine introduces for its own internal use are anonymous, and they should be placed in the next free places of the stack. Within the subroutine, however, we can make distinction between three types of quantities.

The parameters. We use the name "parameters" for all the information that is presented to the subroutine when it is called in by the main program; function arguments, if any, are therefore parameters. The data grouped under the term "link" are also considered as parameters; the link comprises all the data necessary for the continuation of the main program when the subroutine has been completed. Should one wish to do so one can leave the first free place in the stack open, so that the subroutine can place its "function value" there. Thereafter the next places in the stack are used for the parameters. (In some respects it is convenient if all parameters occupy the same number of places in the stack; if, as in ALGOL 60, a parameter may be given in the form of an arbitrarily complicated expression, the limited parameter space in the stack can refer to a point in the memory where further specification of the parameter is given. The amount of information for the link can also be regarded as being constant. Hence the amount of stack space used for the parameters is constant and known for every call.)

The local variables. In general the subroutine itself is a piece of program in which explicit reference is made to a number of quantities introduced by the subroutine. Their values are no longer of interest as soon as the subroutine has been completed. In the course of the execution of the subroutine they can take on a number of different values in succession, and their values can be used more than once (this is the reason why they cannot remain anonymous in the subroutine itself). We allow the amount of memory space occupied by these local variables to be dependent on one or more parameters, e.g. one of the input parameters can be the length of a local vector. We restrict ourselves to the case that the amount of storage space occupied by the local variables becomes known as soon as the input parameters are given, and that it remains constant during that particular activation of the subroutine. (This restriction is in accordance with ALGOL 60; from a logical point of view the restriction is not essential but it simplifies matters considerably.) We therefore know at the beginning of the subroutine, how much memory space the local variables will require this time, and can see to it that they will be stored in the stack immediately after the parameters.

The "most anonymous" intermediate results. During the execution of the subroutine intermediate results that are anonymous even in the text of the subroutine also play a role: for, in general, expressions will have to be evaluated there too, and the subroutine will in its turn call in one or more subroutines itself. These "most anonymous" intermediate results are placed in the stack in the same way as those of the main program, but we must now begin at the first free place following the last local variable.

14.4 Consequences

The main program used the stack exclusively for intermediate numerical results that were formed and added to the stack once, and were later used once and removed. Until then there was no need to store the stack in a random access memory, for our interest was at all times restricted to the youngest element in the stack. In principle we could have used a small magnetic tape that would have to move one place forward in writing and one place backward in reading. The fact that random access is not necessary there, is a direct consequence of the fact that these stack places as such need not be explicitly mentioned in the description of the computation.

Inside the subroutine we store the most anonymous intermediate results in the "top" of the stack in just the same way. Every reference to a local quantity, however, implies that one is interested in a place that is situated deeper within the stack, and *here* one is interested in random access to the stack places, in other words, we must be able to give the places deeper in the stack some kind of address. The point in the stack from which the latter is available for a subroutine, is handed over to the subroutine at the moment it is called in. We assume a static, i.e. constant description of the subroutine to be present in the memory: as a result, this description must be sensitive to the dynamic specification, as described above, of the reference point in the stack. The static reference (static addressing) of the local variables can only occur in terms of a fixed position with respect to the reference point. The value of this reference point is derived from the value of the stack pointer at the moment of the call, and will therefore generally vary from call to call.

14.5 The Link

We must now investigate which data are to be stored in the stack under the heading "link". For the sake of simplicity we regard our computer as consisting of two parts. We refer to the memory space required for the storing of the program and of the stack as "the memory", and we will call the rest "the arithmetic unit". The following considerations—suitably interpreted—apply to both a built-in and a programmed arithmetic unit.

We regard the operation "$v_2 := C$" of our example as an elementary operation of the arithmetic unit. As a result of this operation information in the memory has been modified, but other changes have occured too. Before the execution of this operation, the state of the arithmetic unit was such that the order "$v_2 := C$" was the next to be obeyed, after its execution the state is such that the next order is to be obeyed. Part of the arithmetic unit therefore stores information specifying its state (it contains something equivalent to an order counter). If the text of the program does not specify explicitly the values of the stack pointer, and the arithmetic unit is therefore obliged to keep track of it, the value of the stack pointer is another aspect of the state of the arithmetic unit. One can say quite generally that every instruction is carried out correctly, provided that the arithmetic unit is initially in the appropriate state, and part of the execution of an order is the modification of the state of the arithmetic unit in such a way that the next order will, in its turn, be executed correctly. We now consider the case that the value of C cannot be found ready-made in the memory, but must be calculated by means of an function subroutine. As we are looking for an arrangement in which it is immaterial to the surroundings where C comes from as long as it arrives in the desired stack location, it is necessary that after completion of "$v_2 := C$", the arithmetic unit is left behind in exactly the same state as if C had been transported from the memory by means of an elementary operation. This must hold regardless of the complexity of the subroutine used to calculate C. As the subroutine is in general a piece of program that takes full advantage of the flexibility of the arithmetic unit, its execution obviously implies a series of changes within the arithmetic unit, and we must therefore be willing to record sufficient data regarding the state of the arithmetic unit in order to be able to reconstruct it later. These are the data that must be stored in the stack under the name "link" when a subroutine is called in. (It may be possible that some of the disturbances will be annihilated automatically, e.g. the filling of the stack, which is, in principle at any rate, emptied again. For such "reversible" disturbances the reconstruction requires no extra measures.)

It is clear that some form of "return address" is part of the link data. What the link data should furthermore include depends on the number of different aspects of the state of the arithmetic unit, and whether their changes are intrinsically irreversible. Filling the stack is a reversible process, except when the language permits—as ALGOL 60 does—that a subroutine under execution remains unfinished on account of some criterion, and is left once for all via an exit other than its normal "return". A function subroutine will generally be called in during the evaluation of an expression, so that some stack places are filled with anonymous intermediate results that will never be used on account of the unusual exit from the function routine. In that case one must be able to reconstruct up to which point the stack contents are still of interest. This facility requires an additional record in the link.

Furthermore there exists a certain hierarchy in the information specifying the state of the arithmetic unit. If we regard the state of the arithmetic unit between two orders, there is certain information (order counter) that sees to the transition to the next order. During the execution of an order, however, we can distinguish between different “sub-states”: the information specifying these substates is of no significance at the moment of transition from one order to the next, and there is therefore no need to record this information in the link, as long as the subroutine mechanism only operates between orders. The specification of the substates should be recorded in the link, however, as soon as one allows the subroutine mechanism to operate during the course of an elementary operation, i.e. when (for the execution of one of its sub-operations) a subroutine must be called in that may possibly make use of the full flexibility of the arithmetic unit.

When a subroutine, say subroutine *A*, is activated, it must operate in the stack starting at a point that only becomes known at the moment of call. The parameters (including the link) and the local variables of *A* have a fixed positioning with respect to each other, but their position as a whole is only determined at the call: the quantity specifying the position of the whole block, is called a *parameter pointer.* The program for subroutine *A* can only be executed correctly if the current value of the parameter pointer is at the disposal of the arithmetic unit, in other words, the parameter pointer can be regarded as one of the aspects of the state of the arithmetic unit, and it must therefore be recorded amongst the link data. Now consider the case that subroutine *A* calls in subroutine *B.* We call the value of the parameter pointer indicating the parameters and local variables of *A* and *B* respectively, *P P A* and *P P B* respectively. If subroutine *B* is called in by *A,* the value *P P A* is recorded as part of the link formed at this call. The place in the stack where this link is recorded is deduced from the value of the stack pointer at that moment and is, by definition, recorded in the parameter pointer, which now assumes the value we called *P P B.* When we return from *B to A* the return address is found in the stack under control of *P P B*, as well as the new value *P P A* of the parameter pointer; finally, at the operation “return”, the new value of the stack pointer, which suddenly decreases, can be deduced from the old value of the parameter pointer (in our case from *P P B*)

In this process *nothing* forbids *A* from being identical with *B.* The subroutine only has to appear in the memory once, but it may then have more than one simultaneous “incarnation” from a dynamic point of view: the “innermost” activation causes the same piece of text to work in a higher part of the stack. Thus the subroutine has developed into a defining element that can be used completely recursively.

14.6 Recursive Techniques and ALGOL 60

Every procedure is regarded as a subroutine in the sense described above. For the sake of simplicity a block that is not a procedure can also be treated as a subroutine, be it that this subroutine is only called in at one point.

One of the complications of ALGOL 60 is that not only local variables may be used in each block, but that explicit reference may be made in every block to variables that are local in a lexicographically enclosing block. When a subroutine is called in, the link contains *two* parameter pointer values for this purpose. Firstly, the youngest parameter pointer value corresponding to the block in which the *call* occurs (as described in the previous section), secondly, the value of the parameter pointer corresponding to the most recent, not yet completed, activation of the first block that lexicographically encloses the *block* of the subroutine called in. As already mentioned, the first parameter pointer value plays a vital role in the return at the end of the subroutine, the second is indispensable in localizing the global variables in the stack. As the second parameter pointer, by definition, points to a link in the stack, which in its turn contains a second parameter pointer value corresponding to the next enclosing block, the arithmetic unit can trace this "chain" and, in doing so, will find all parameter pointer values that may be necessary for localizing any global variable in which it may be interested.

One can assign a so-called *block number* to each block, indicating the number of blocks which enclose it lexicographically: the main program therefore has a block number = 0. If the program refers to a global variable it is obviously necessary to specify the block in which the global variable was declared; the block number serves this purpose and, under control of this block number, the arithmetic unit can find the parameter pointer value it now needs. Further, the reference to a global variable will specify which variable of this block is required: its position with respect to the parameter pointer value just found achieves this. The introduction of the block numbers also makes it possible that the arithmetic unit has immediate access to all the parameter pointer values it may need. They can be stored in order of increasing block number in a so-called "display" (e.g. a series of index registers, numbered 0, 1, 2, 3, ... etc., as many as the maximum value of the block number). By tracing the second chain of parameter pointers one can, when necessary (amongst others at the return, in the call of a formal procedure, and at the beginning of the evaluation of a non-trivial formal parameter), bring the display up to date. This needs—and can in general—only be done for block numbers not exceeding the number of the block we are about to enter. It will not surprise the reader that the block number is to be regarded as specifying the state of the arithmetic unit; in consequence it has to be stored amongst the link data in the stack.

Acknowledgements

The author is especially indebted to Professor A. van Wijngaarden and J. A. Zonneveld, co-members of the team that is engaged in the construction of a compiler for ALGOL 60 at the moment of writing. During the course of this work many aspects, which he had tentatively written down at an earlier stage, became clearer, and their consequences became apparent under the pressure of the circumstances.

Furthermore, the author had the great privilege of having some inspiring conversations on this subject with Professor H. D. Huskey, when he was the guest of the Mathematical Centre, Amsterdam, in the summer of 1959.

Reference

[1] Bauer, F. L., and K. Samelson: Sequentielle Formelübersetzung. Elektronische Rechenanlagen **1**, H. 4, 176–182 (1959).

Mathematical Centre
Amsterdam-O/Niederlande
2e Boerhaavestraat 49

(Received Mai 11, 1960)

Druck der Universitätsdruckerei H. Stürtz AG., Würzburg

15

Some Meditations on Advanced Programming

E. W. Dijkstra

In case you expect me to give a complete, well-balanced and neutral survey of the advanced programming activities of the world, I must warn you that I do not feel inclined, nor entitled to do so.

My title already indicates that I am going to meditate on the subject, which is something quite different from giving a survey. Perhaps the title of my paper would have been more outspoken if it had been "My meditations on Advanced Programming" for I intend to present a picture in the way I wish to see it; and I should like to do so in all honesty without any claim to objectivity. I intend to do so because I have a feeling that I serve you better by giving you an honest personal conviction than by presenting you with the colourless average of conflicting current opinions of other people.

You will observe that I fail to give you a generally acceptable definition of the subject "Advanced Programming". I think that in my own appreciation of the subject the description "Advancing Programming" would have been a better qualification. I like many activities which are worthy, I think, of the name "Advanced Programming" but I do not like these activities so much for the sake of their output, the programs that have resulted from them, as for what these activities can teach us. If I am willing to study them, to meditate upon them, I am willing to do

Originally published as Edsger W. Dijkstra, Some Meditations on Advanced Programming, Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27-September 1, 1962, (North-Holland, 1962), 535–38.
Republished with permission.
This chapter was produced using the transcription https://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD32.html produced by Tristram Brelstaff and was checked against the original text.

so in the hope this study or these meditations will give me a clearer understanding of the programmer's task, of his ends and his means. Therefore I should like to draw your attention in particular to those efforts and considerations which try to improve "the state of the Art" of programming, maybe to such an extent that at some time in the future we may speak of "the state of the Science of Programming".

And a look around us will convince us that this improvement is very urgent, for on the whole the programmers' world is a very dark one with only just the first patches of a brighter sky appearing at the horizon. For the present-day darkness in the programmers' world the programmers themselves are responsible and nobody else. But before we put too much blame on them, let us look for a moment how their world came into existence.

When the first automatic electronic computers started to work more or less properly, mankind was faced with a new technical wonder, with a most impressive achievement of technical skill, and as a result, everybody was highly impressed and rightly so. Under these circumstances it was completely natural that the structure of these early machines was mainly decided by the technical possibilities at that time. And under these circumstances it would have been an undreamt-of undecency for programmers to dare to suggest that those clever designers had not at all built the machines that programmers wanted. Therefore this thought hardly entered the programmers' minds. On the contrary: faced on the one hand with the new computers and on the other hand heaps of problems waiting for their solution, they have done their utmost best to accomplish the task with the equipment that had become available. They have accepted the full challenge. The potentialities of the computers have been exhausted to slightly beyond their utmost limits, the nearly impossible jobs have yet been done by using the machines in all kinds of curious and tricky ways, which were completely unintended and not even foreseen by the designers. In this atmosphere of pioneering, programming has arisen not as a science but as a craft, as an occupation where man, under the pressure of the circumstances was guided more by opportunism than by sound principles. This—I should like to call it "unhygienic"—creativity and shrewdness of the programmers has had a very bad influence on machine designers; for after some time they felt free to include all kinds of curious facilities of doubtful usability, reassuring themselves by their experience that, no matter how crazy a facility they provided, always a more crazy programmer would emerge that would manage to turn it into something profitable—as if this were sufficient justification for its inclusion.

In the meantime programming established itself as a discipline where on the whole standards for quality were extremely crude and primitive. The main—and often only—possible virtues of a program were its quantitative characteristics, viz. its speed and its storage requirements. Space and time became the exclusive aspects of efficiency. And in various places these standards are still in full vigour: not so long ago I heard of two cases, one where a machine was not bought because its multiplication speed was too low—and this may be a valid argument—and another case where a certain machine was selected because its multiplication was so fast. And this last decision was taken without the validity of this criterion being questioned.

Apart from the programs that have been produced, the programmers' contribution to human knowledge has been fairly useless. They have concocted thousands and thousands of ingenious tricks but they have given this chaotic contribution without a mechanism to appreciate, to evaluate these tricks, to sort them out. And as many of these tricks could only be played by virtue of some special property of some special machine their value was rather volatile. But the tricks were defended in the name of the semi-god "Efficiency" and for a long time there was hardly an inkling that there could be anything wrong with tricks. The programmer was judged by his ability to invent and his willingness to apply tricks. This opinion is still a wide-spread phenomenon: in advertisements asking for programmers and in psychological tests for this job it is often required that the man should be "puzzle-minded"; this in strong contrast to the opinion of the slowly growing group of people who think it more valuable that the man should have a clear and systematic mind.

But, as I told you, the sky above the programmer's world is brightening slowly. Before I am going to draw your attention to some discoveries that are responsible for this improvement I should like to state as my opinion that it is relatively unimportant whether these discoveries are really new discoveries or whether they are rediscoveries of things perfectly well known to people like, say, Turing or von Neumann. For in the latter case the important and new thing is that a greater number of people become aware of such a fact and that a greater number of people realize that these considerations are not just theoretical considerations but that they may have tangible, practical results. In this light one might feel inclined to summarize the achievements of advanced programming as some purely educational successes: "At last programmers have started to educate one another to at least some extent." I shall not protest against this summary provided one agrees with my opinion that mutual education is one of the major difficult tasks of mankind.

One important rediscovery is that of the well-known equivalence of designing a machine and making a program. At this moment one might well ask oneself why I ask attention for such a well-known fact. I have very good reasons to do so, for it has a great potential influence which is often overlooked: it enables the man that regards himself as a programmer to contribute to the field that is generally regarded as “machine design”, and this is a very fortunate circumstance.

Some fifteen or ten years ago the design and construction of a new, unique computer was a well-established and respectable occupation for UNIVERSITY LABORATORIES. Many of these “laboratory machines” were, each in their own private way, revolutionary contraptions. From then onwards this custom died out and design and construction of automatic computers became more and more an exclusively industrial activity. Five years ago most of us felt this as a perfectly natural development: construction of new computers became an extremely costly affair and it was generally felt that the time had come to leave this activity to the specialized industries. Now, five years later, we can only regret this development, for the computers on the market today are, on the whole, very disappointing. Certainly, they are faster, they are much more reliable than the old laboratory machines, but, on the other hand, they are often boring, uninspiring and hopelessly old-fashioned as well. For instance, the commercial requirement that all the programs made for some older machine from the same manufacturer should, without any modification, be acceptable to the new machine has led to the design of new machines of which the order codes include the order code of the previous one in its entirety. Such a policy, however is a never failing mechanism to prolong the lifetime of previous mistakes. Some time ago we were offered the slogans about “the computers of the second generation”, but to my taste many of them were as dull as their parents. Apparently a nice computer has at least one property in common with a gentleman, viz. that it takes at least three generations to produce one! Most of the industries, particularly the bigger ones, proved to be very conservative and reactionary. They seem to design for the customer that believes the salesman who tells him that machine so-and-so is just the machine he wants. But the poor customer who happened to know already, all by himself, what he wants is often forced to accept a machine with which he is already disgusted before the thing is installed in his establishment. Under the present circumstances, it is, commercially speaking, apparently not too attractive to put a nice computer on the market. This is a sorry state of affairs, many a programmer suffers regularly from the monstrosity of his tool and we can only hope for a better future with nicer machines. In the meantime he can program; taking some efficiency considerations for granted he can force his machine to behave as he wishes: when making a programming system he designs a machine as it should have been. Thanks to the logical equivalence between designing a machine and making a program, programmers can contribute to future machine design, by exploring on paper, in software, the possibilities of machines with a more revolutionary structure.

The equivalence of making a program and designing a machine has another, may be far-reaching consequence of a much more practical nature. It is not unusual to regard a classical computer as a sequential computer coupled to a number of communication mechanisms for input and output. Such a communication mechanism, however, performs in itself a sequential process—usually of a cyclic nature, but that feature is of no importance now. For this reason we can regard a classical machine, communication mechanism included, as a group of loosely connected sequential machines, with interlocks, where necessary, to prevent them to get too much out of phase with one another. The next step is to use the central computer not for only one sequential process, but to equip it with the possibility to divide its attention between an arbitrary number of such loosely connected sequential processes. One can do so with complete preservation of the symmetry between the sequential processes, to which a distinct piece of hardware corresponds on the one hand and those taken care of by the central computer on the other hand, or even by one of the central processors, as the case may be. The difference between a modest and an ambitious installation may be that a couple of sequential processes, that in the modest installation are performed by the central computer, are performed by private hardware in the ambitious installation. But the above-mentioned equivalence between designing a machine and making a program, between performing a process either by hardware or by software, should be exploited to guarantee that the program acceptable for the one installation is also acceptable for the other. The above considerations are important because a machine rigorously designed along the above lines would greatly facilitate the manufacturer's task of equipping his product with the required software. The moral of this is that, if at the present moment many manufacturers have great difficulties in fulfilling their software obligations and if one of the main sources of their trouble is that no two installations of the same machine are identical, their trouble could very well be a self-inflicted pain.

In this connection I should like to mention that I am fully aware of the fact that my previous picture of the commercial computer market was somewhat one-sided. Many of you will realize that at least one of the commercial products shows a great number of the "nice properties" just mentioned. In my opinion, this particular computer should be regarded as one of the brightest patches in the sky.

Now I want to turn my attention to one of the most important facts that happened in the programmers' world since the UNESCO Conference in 1959, viz. the publication of the famous "Report on the Algorithmic Language ALGOL 60", edited by Dr P. Naur. I shall not discuss here the merits of the language ALGOL 60, nor shall I go into the question whether it has achieved its original aims or not. I intend to restrict myself to a discussion of the consequences of this publication, and of the influence it has had in the world of programming; for this influence has been tremendous. Briefly, I could formulate it as follows: "Through its merits ALGOL 60 has inspired a great number of people to make translators for it, through its defects it has induced a great number of people to think about the aims of a "Programming Language"." ALGOL 60, in all probability and in accordance with the wish of its authors, will be superseded by some better language in due time, but for much, much longer we shall be able to trace its educational effects.

Programming language, translator and computer, these three together for a tool, and in thinking about this tool as a whole, new dimensions have been added to the old concept of "reliability". In connection with the third of the three components, viz. the computer, concern about its reliability is as old as computers themselves; the acceptance test is a well-known phenomenon.

But what is the value of such an acceptance test? It is certainly no guarantee that the machine is correct—that the machine acts according to its specifications. It only says that in these specific test programs the machine has worked correctly. If the design is based on some critical assumption, we can only conclude, that in these test programs the corresponding critical situations apparently did not arise. If the design still contains errors, we can conclude, that in these specific test programs these logical errors apparently did not matter. But as users, we are not interested in the test programs, we are interested in our own programs, and from the successful acceptance test we should like to conclude that the machine works correctly in our programs also! But we cannot draw this conclusion.

The best thing that a successful acceptance test can do is to strengthen our belief in the machine's correctness, to increase the plausibility that it will perform any program in accordance with the specifications. The basic property of the user's program is that it will certainly require from the machine to perform actions it has never done before. Machine designers have seen this difficulty quite clearly. They have realized that the successful acceptance test has only value as far as future programs are concerned, provided the actions performed in the test programs can be regarded as representative for all its possible operations, and they can only be representative by virtue of the clean and systematic structure of the machine itself. The above is common knowledge among machine designers; curiously enough, this is not true for translator makers, for whose activity the same considerations apply.

In order that the tool, consisting of programming language, translator and machine, be a reliable tool, it is, of course, mandatory that all its components be reliable. One should expect that the translator maker, who in contrast to the machine designer has to deal with logical errors only, should do his job at least as well as the machine builder. But I am afraid that the converse is true. At the

Rome Conference early in 1962, I was surprised to hear that the extensive translators for symbolic languages constructed in the United States continued to show up errors for years. I was shocked, however, when I saw the fatalistic mood in which this sorry state was accepted as the most natural thing in the world. This same attitude is reflected in the terms of reference of an ISO committee which deals with the standardization of programming languages: there one finds the recommendation to construct for any standard language a set of standard test examples on which any new translator for such a language could be tried out. But one finds no hint that the correct processing of these standard test examples obviously is only a trivial minimum requirement, no trace of the consideration that our belief in the correctness of a translator can never be founded on successful tests alone, but is ultimately derived from the clean and systematic structure of the translator and from nothing else. In deciding between reliability of the translation process on the one hand, and the production of an efficient object program on the other hand, the choice often has been decided in favour of the latter. But I have the impression that, the pendulum is now swinging backwards.

For instance: if one gets a much more powerful machine in one's establishment than the one one had before, one can react to this in two different ways. The classical reaction is that the new machine is so much more expensive, that it is even more mandatory that no expensive computing time of the new machine should be wasted, that the new machine should be used as effectively as possible, etc., etc. On the other hand one can also reason as follows: as the new machine is much faster, time does not matter so much any more; as in the new computer the cost per operation is less than in the previous one, it becomes more realistic to investigate whether we can invest some of the machine's speed in other things than in sheer production, say in convenience for the user—what we do already when we use a convenient programming language—or in elegance and reliability of the translator, thus increasing the quality of our output.

Also it is more widely recognised now than a couple of years ago that the construction of an optimizing translator is, essentially, a nasty job. Optimizing means improving the object program, i.e. making a more efficient object program than the one produced by straightforward, but reliable and trustworthy translation techniques. Optimization means "taking advantage of a special situation". If one optimizes in one respect, it is not an impossible burden to verify that the shortcut introduced in the object program does not lead to undesired results. If, however, one optimizes in two different respects, the duty of verification becomes much harder, for one has to verify not only that the two methods are correct in themselves, but one must also check that they do not interfere with one another. If one optimizes in more different respects, the task to create confidence in the correctness of the translator explodes exponentially. As a result it is no longer possible to recommend a computer by pointing to, say, the size of the translators available for it. On the contrary, the more extensive and shrewd a translator is, the more doubtful is its quality. Further, for the necessity of such extensive optimization efforts one might, finally, blame the computer in question: if one really needs such an intricate process as an optimizing translation to load one's programs, one feels inclined to defend the opinion that, apparently, the computer is not too well suited for its task. In short, the construction of intricate optimizing translators is an act, the wisdom of which is subject to doubt, and there is certainly a virtue in efforts to remove the need for them, e.g. the design of computers where these optimization tricks do not pay, or at least do not pay so much.

With regard to the structure of a translator, ALGOL 60 has acted as a great promoter of non-optimizing translators. The fact is that the language as it stands is certainly not an open invitation for optimization efforts. For those who thought they knew how to write optimizing translators—be it for less flexible languages—this has been one of the reasons to reject ALGOL 60 as a serious tool. In my opinion these people bet on the wrong horse. I do not agree with them although I can sympathize with them; if one has solved a problem one tends to get attached to it and if one likes one's solution for it, it is, of course, a little bit hard to switch over to an attitude in which the problem is not considered worth solving any more. The experience with ALGOL 60 translation has taught us still another thing. Some translator makers could not refrain from optimizing, but finding the task as such too difficult to do they tried to ease matters by introducing additional restrictions into the language. The fact that their translators had only to deal with a restricted language, however, did not speed up the translator construction; the task to exploit the restrictions to full advantage has prevented this.

Thus we have arrived at the third component of our tool, viz. the language; and the language also should be a reliable one. In other words it should assist the programmer as much as possible in the most difficult aspect of his task, viz. to convince himself—and those others who are really interested—that the program he has written down defines in fact the process he wanted to define. Obviously the language rules may not contain traps of the kind of which there are still some in ALGOL 60, where, for instance, '**real array**' may be abbreviated into '**array**', but '**own real array**' may not be abbreviated into '**own array**'. The next obvious requirement is that those rules which define a legal text do not leave any doubt as to whether a given text is legal or not, e.g., if there should be a restriction with respect to recursive use of a procedure, it should be clear under what conditions these restrictions apply, in particular when the term "recursive use" applies. I mention this particular example because here it is by no means obvious. Finally, when faced with an undoubtedly legal text we want to be quite sure what it means. This implies that the semantic definition should be as rigorous as possible. In short, we need a complete and unambiguous pragmatic definition of the language, stating explicitly how to react to any text. So much for the necessity that the tool be reliable.

As my very last remark I should like to stress that the tool as a whole should have still another quality. This is a much more subtle one; whether we appreciate it or not depends much more on our personal taste and education, and I shall not even try to define it. The tool should be charming, it should be elegant, it should be worthy of our love. This is no joke, I am terribly serious about this. In this respect the programmer does not differ from any other craftsman: unless he loves his tools it is highly improbable that he will ever create something of superior quality.

Thus, at the same time these considerations tell us the greatest virtues a program can show: Elegance and Beauty.

16

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

The Problem

To begin, consider *N* computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these *N* cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each

Originally published in Communications of the ACM, Volume 8 / Number 9 / September, 1965.

other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the *N* computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the *N* computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

The Solution

The common store consists of:

"**Boolean array** b, $c[1{:}N]$; **integer** k"

The integer k will satisfy $1 \leq k \leq N$, $b[i]$ and $c[i]$ will only be set by the ith computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of k is immaterial.

The program for the ith computer ($1 \leq i \leq N$) is:

```
"integer j;
Li0:   b[i] := false;
Li1:   if $k \neq i$ then
Li2:   begin c[i] := true;
Li3:   if b[k] then k := i;
       go to Li1
       end
          else
Li4:   begin c[i] := false;
          for j := 1 step 1 until N do
             if $j \neq i$ and not c[j] then go to Li1
       end;
       critical section;
       c[i] := true; b[i] := true;
       remainder of the cycle in which stopping is allowed;
       go to Li0"
```

The Proof

We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement *Li*4 without jumping back to *Li*1, i.e., finding all other c's **true** after having set its own c to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to *Li*1) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the kth computer is not among the looping ones, $b[k]$ will be **true** and the looping ones will all find $k \neq i$. As a result one or more of them will find in *Li*3 the Boolean $b[k]$ **true** and therefore one or more will decide to assign "$k := i$". After the first assignment "$k := i$", $b[k]$ becomes **false** and no new computers can decide again to assign a new value to k. When all decided assignments to k have been performed, k will point to one of the looping computers and will not change its value for the time being, i.e., until $b[k]$ becomes **true**, viz., until the kth computer has completed its critical section. As soon as the value of k does not change any more, the kth computer will wait (via the compound statement *Li*4) until all other c's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their c **true**, as they will find $k \neq i$. And this, the author believes, completes the proof.

17

Go To Statement Considered Harmful

E. W. Dijkstra

Key Words and Phrases

go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

***CR* Categories**

4.22, 5.23, 5.24

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

Originally published in Communications of the ACM, Volume 11 / Number 3 / March, 1968.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of (*A*1, *A*2, ··· , *An*)), or conditional expressions as introduced by J. McCarthy (*B*1 ⟶ *E*1, *B*2 ⟶ *E*2, ··· , *Bn* ⟶ *En*), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while *B* repeat *A*** or **repeat *A* until *B***). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called “dynamic index,” inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer’s control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one’s program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Giuseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

References

1. WIRTH, NIKLAUS, and HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, and JACOPINI, GIUSEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

Edsger W. Dijkstra
Technological University
Eindhoven, The Netherlands

18 The Structure of the "THE"-Multiprogramming System

Edsger W. Dijkstra

A multiprogramming system is described in which all activities are divided over a number of sequential processes. These sequential processes are placed at various hierarchical levels, in each of which one or more independent abstractions have been implemented. The hierarchical structure proved to be vital for the verification of the logical soundness of the design and the correctness of its implementation.

KEY WORDS AND PHRASES

operating system, multiprogramming system, system hierarchy, system structure, real-time debugging, program verification, synchronizing primitives, cooperating sequential processes, system levels, input-output buffering, multiprogramming, processor sharing, multiprocessing

CR CATEGORIES

4.30, 4.32

18.1 Introduction

In response to a call explicitly asking for papers "on timely research and development efforts," I present a progress report on the multiprogramming effort at the Department of Mathematics at the Technological University in Eindhoven.

Presented at an ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1–4, 1967.
Originally published in Communications of the ACM, Volume 11 / Number 5 / May, 1968.

Having very limited resources (viz. a group of six people of, on the average, half-time availability) and wishing to contribute to the art of system design—including all the stages of conception, construction, and verification, we were faced with the problem of how to get the necessary experience. To solve this problem we adopted the following three guiding principles:

(1) Select a project as advanced as you can conceive, as ambitious as you can justify, in the hope that routine work can be kept to a minimum; hold out against all pressure to incorporate such system expansions that would only result into a purely quantitative increase of the total amount of work to be done.

(2) Select a machine with sound basic characteristics (e.g. an interrupt system to fall in love with is certainly an inspiring feature); from then on try to keep the specific properties of the configuration for which you are preparing the system out of your considerations as long as possible.

(3) Be aware of the fact that experience does by no means automatically lead to wisdom and understanding; in other words, make a conscious effort to learn as much as possible from your previous experiences.

Accordingly, I shall try to go beyond just reporting what we have done and how, and I shall try to formulate as well what we have learned.

I should like to end the introduction with two short remarks on working conditions, which I make for the sake of completeness. I shall not stress these points any further.

One remark is that production speed is severely slowed down if one works with half-time people who have other obligations as well. This is at least a factor of four; probably it is worse. The people themselves lose time and energy in switching over; the group as a whole loses decision speed as discussions, when needed, have often to be postponed until all people concerned are available.

The other remark is that the members of the group (mostly mathematicians) have previously enjoyed as good students a university training of five to eight years and are of Master's or Ph.D. level. I mention this explicitly because at least in my country the intellectual level needed for system design is in general grossly underestimated. I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expense.

18.2 The Tool and the Goal

The system has been designed for a Dutch machine, the EL X8 (N.V. Electrologica, Rijswijk (ZH)). Characteristics of our configuration are:

(1) core memory cycle time 2.5μsec, 27 bits; at present 32K;

(2) drum of 512K words, 1024 words per track, rev. time 40msec;

(3) an indirect addressing mechanism very well suited for stack implementation;

(4) a sound system for commanding peripherals and controlling of interrupts;

(5) a potentially great number of low capacity channels; ten of them are used (3 paper tape readers at 1000char/sec; 3 paper tape punches at 150char/sec; 2 teleprinters; a plotter; a line printer);

(6) absence of a number of not unusual, awkward features.

The primary goal of the system is to process smoothly a continuous flow of user programs as a service to the University. A multiprogramming system has been chosen with the following objectives in mind: (1) a reduction of turn-around time for programs of short duration, (2) economic use of peripheral devices, (3) automatic control of backing store to be combined with economic use of the central processor, and (4) the economic feasibility to use the machine for those applications for which only the flexibility of a general purpose computer is needed, but (as a rule) not the capacity nor the processing power.

The system is not intended as a multiaccess system. There is no common data base via which independent users can communicate with each other: they only share the configuration and a procedure library (that includes a translator for ALGOL 60 extended with complex numbers). The system does not cater for user programs written in machine language.

Compared with larger efforts one can state that quantitatively speaking the goals have been set as modest as the equipment and our other resources. Qualitatively speaking, I am afraid, we became more and more immodest as the work progressed.

18.3 A Progress Report

We have made some minor mistakes of the usual type (such as paying too much attention to eliminating what was not the real bottleneck) and two major ones.

Our first major mistake was that for too long a time we confined our attention to “a perfect installation”; by the time we considered how to make the best of it, one of the peripherals broke down, we were faced with nasty problems. Taking care of the "pathology" took more energy than we had expected, and some of our troubles were a direct consequence of our earlier ingenuity, i.e. the complexity of the situation into which the system could have maneuvered itself. Had we paid attention to the pathology at an earlier stage of the design, our management rules would certainly have been less refined.

The second major mistake has been that we conceived and programmed the major part of the system without giving more than scanty thought to the problem of debugging it. I must decline all credit for the fact that this mistake had no serious consequences—on the contrary! one might argue as an afterthought.

As captain of the crew I had had extensive experience (dating back to 1958) in making basic software dealing with real-time interrupts, and I knew by bitter experience that as a result of the irreproducibility of the interrupt moments a program error could present itself misleadingly like an occasional machine malfunctioning. As a result I was terribly afraid. Having fears regarding the possibility of debugging, we decided to be as careful as possible and, prevention being better than cure, to try to prevent nasty bugs from entering the construction.

This decision, inspired by fear, is at the bottom of what I regard as the group's main contribution to the art of system design. We have found that it is possible to design a refined multiprogramming system in such a way that its logical soundness can be proved a priori and its implementation can admit exhaustive testing. The only errors that showed up during testing were trivial coding errors (occurring with a density of one error per 500 instructions), each of them located within 10 minutes (classical) inspection by the machine and each of them correspondingly easy to remedy. At the time this was written the testing had not yet been completed, but the resulting system is guaranteed to be flawless. When the system is delivered we shall not live in the perpetual fear that a system derailment may still occur in an unlikely situation, such as might result from an unhappy "coincidence" of two or more critical occurrences, for we shall have proved the correctness of the system with a rigor and explicitness that is unusual for the great majority of mathematical proofs.

18.4 A Survey of the System Structure

Storage Allocation. In the classical von Neumann machine, information is identified by the address of the memory location containing the information. When we started to think about the automatic control of secondary storage we were familiar with a system (viz. GIER ALGOL) in which all information was identified by its drum address (as in the classical von Neumann machine) and in which the function of the core memory was nothing more than to make the information “page-wise” accessible.

We have followed another approach and, as it turned out, to great advantage. In our terminology we made a strict distinction between memory units (we called them “pages” and had “core pages” and “drum pages”) and corresponding information units (for lack of a better word we called them “segments”), a segment just fitting in a page. For segments we created a completely independent identification mechanism in which the number of possible segment identifiers is much larger than the total number of pages in primary and secondary store. The segment identifier gives fast access to a so-called “segment variable” in core whose value denotes whether the segment is still empty or not, and if not empty, in which page (or pages) it can be found.

As a consequence of this approach, if a segment of information, residing in a core page, has to be dumped onto the drum in order to make the core page available for other use, there is no need to return the segment to the same drum page from which it originally came. In fact, this freedom is exploited: among the free drum pages the one with minimum latency time is selected.

A next consequence is the total absence of a drum allocation problem: there is not the slightest reason why, say, a program should occupy consecutive drum pages. In a multiprogramming environment this is very convenient.

Processor Allocation. We have given full recognition to the fact that in a single sequential process (such as can be performed by a sequential automaton) only the time succession of the various states has a logical meaning, but not the actual speed with which the sequential process is performed. Therefore we have arranged the whole system as a society of sequential processes, progressing with undefined speed ratios. To each user program accepted by the system corresponds a sequential process, to each input peripheral corresponds a sequential process (buffering input streams in synchronism with the execution of the input commands), to each output peripheral corresponds a sequential process (unbuffering output streams in synchronism with the execution of the output commands); furthermore, we have the “segment controller” associated with the drum and the “message interpreter” associated with the console keyboard.

This enabled us to design the whole system in terms of these abstract “sequential processes.” Their harmonious cooperation is regulated by means of explicit mutual synchronization statements. On the one hand, this explicit mutual synchronization is necessary, as we do not make any assumption about speed ratios; on the other hand, this mutual synchronization is possible because “delaying the progress of a process temporarily” can never be harmful to the interior logic of the process delayed. The fundamental consequence of this approach—viz. the explicit mutual synchronization—is that the harmonious cooperation of a set of such sequential processes can be established by discrete reasoning; as a further consequence the whole harmonious society of cooperating sequential processes is independent of the actual number of processors available to carry out these processes, provided the processors available can switch from process to process.

System Hierarchy. The total system admits a strict hierarchical structure.

At level 0 we find the responsibility for processor allocation to one of the processes whose dynamic progress is logically permissible (i.e. in view of the explicit mutual synchronization). At this level the interrupt of the real-time clock is processed and introduced to prevent any process to monopolize processing power. At this level a priority rule is incorporated to achieve quick response of the system where this is needed. Our first abstraction has been achieved; above level 0 the number of processors actually shared is no longer relevant. At higher levels we find the activity of the different sequential processes, the actual processor that had lost its identity having disappeared from the picture.

At level 1 we have the so-called "segment controller," a sequential process synchronized with respect to the drum interrupt and the sequential processes on higher levels. At level 1 we find the responsibility to cater to the book-keeping resulting from the automatic backing store. At this level our next abstraction has been achieved; at all higher levels identification of information takes place in terms of segments, the actual storage pages that had lost their identity having disappeared from the picture.

At level 2 we find the "message interpreter" taking care of the allocation of the console keyboard via which conversations between the operator and any of the higher level processes can be carried out. The message interpreter works in close synchronism with the operator. When the operator presses a key, a character is sent to the machine together with an interrupt signal to announce the next keyboard character, whereas the actual printing is done through an output command generated by the machine under control of the message interpreter. (As far as the hardware is concerned the console teleprinter is regarded as two independent peripherals: an input keyboard and an output printer.) If one of the processes opens a conversation, it identifies itself in the opening sentence of the conversation for the benefit of the operator. If, however, the operator opens a conversation, he must identify the process he is addressing, in the opening sentence of the conversation, i.e. this opening sentence must be interpreted before it is known to which of the processes the conversation is addressed! Here lies the logical reason for the introduction of a separate sequential process for the console teleprinter, a reason that is reflected in its name, “message interpreter.”

Above level 2 it is as if each process had its private conversational console. The fact that they share the same physical console is translated into a resource restriction of the form “only one conversation at a time,” a restriction that is satisfied via mutual synchronization. At this level the next abstraction has been implemented; at higher levels the actual console teleprinter loses its identity. (If the message interpreter had not been on a higher level than the segment controller, then the only way to implement it would have been to make a permanent reservation in core for it; as the conversational vocabulary might become large (as soon as our operators wish to be addressed in fancy messages), this would result in too heavy a permanent demand upon core storage. Therefore, the vocabulary in which the messages are expressed is stored on segments, i.e. as information units that can reside on the drum as well. For this reason the message interpreter is one level higher than the segment controller.)

At level 3 we find the sequential processes associated with buffering of input streams and unbuffering of output streams. At this level the next abstraction is effected, viz. the abstraction of the actual peripherals used that are allocated at this level to the “logical communication units” in terms of which are worked in the still higher levels. The sequential processes associated with the peripherals are of a level above the message interpreter, because they must be able to converse with the operator (e.g. in the case of detected malfunctioning). The limited number of peripherals again acts as a resource restriction for the processes at higher levels to be satisfied by, mutual synchronization between them.

At level 4 we find the independent-user programs and at level 5 the operator (not implemented by us).

The system structure has been described at length in order to make the next section intelligible.

18.5 Design Experience

The conception stage took a long time. During that period of time the concepts have been born in terms of which we sketched the system in the previous section. Furthermore, we learned the art of reasoning by which we could deduce from our requirements the way in which the processes should influence each other by their mutual synchronization so that these requirements would be met. (The requirements being that no information can be used before it has been produced, that no peripheral can be set to two tasks simultaneously, etc.). Finally we learned the art of reasoning by which we could prove that the society composed of processes thus mutually synchronized by each other would indeed in its time behavior satisfy all requirements.

The construction stage has been rather traditional, perhaps even old-fashioned, that is, plain machine code. Reprogramming on account of a change of specifications has been rare, a circumstance that must have contributed greatly to the feasibility of the "steam method." That the first two stages took more time than planned was somewhat compensated by a delay in the delivery of the machine.

In the verification stage we had the machine, during short shots, completely at our disposal; these were shots during which we worked with a virgin machine without any software aids for debugging. Starting at level 0 the system was tested, each time adding (a portion of) the next level only after the previous level had been thoroughly tested. Each test shot itself contained, on top of the (partial) system to be tested, a number of testing processes with a double function. First, they had to force the system into all different relevant states; second, they had to verify that the system continued to react according to specification.

I shall not deny that the construction of these testing programs has been a major intellectual effort: to convince oneself that one has not overlooked "a relevant state" and to convince oneself that the testing programs generate them all is no simple matter. The encouraging thing is that (as far as we know!) it could be done.

This fact was one of the happy consequences of the hierarchical structure.

Testing level 0 (the real-time clock and processor allocation) implied a number of testing sequential processes on top of it, inspecting together that under all circumstances processor time was divided among them according to the rules. This being established, sequential processes as such were implemented.

Testing the segment controller at level 1 meant that all "relevant states" could be formulated in terms of sequential processes making (in various combinations) demands on core pages, situations that could be provoked by explicit synchronization among the testing programs. At this stage the existence of the real-time clock—although interrupting all the time—was so immaterial that one of the testers indeed forgot its existence!

By that time we had implemented the correct reaction upon the (mutually unsynchronized) interrupts from the real-time clock and the drum. If we had not introduced the separate levels 0 and 1, and if we had not created a terminology (viz. that of the rather abstract sequential processes) in which the existence of the clock interrupt could be discarded, but had instead tried in a nonhierarchical construction, to make the central processor react directly upon any weird time succession of these two interrupts, the number of "relevant states" would have exploded to such a height that exhaustive testing would have been an illusion. (Apart from that it is doubtful whether we would have had the means to generate them all, drum and clock speed being outside our control.)

For the sake of completeness I must mention a further happy consequence. As stated before, above level 1, core and drum pages have lost their identity, and buffering of input and output streams (at level 3) therefore occurs in terms of segments. While testing at level 2 or 3 the drum channel hardware broke down for some time, but testing proceeded by restricting the number of segments to the number that could be held in core. If building up the line printer output streams had been implemented as "dumping onto the drum" and the actual printing as "printing from the drum," this advantage would have been denied to us.

18.6 Conclusion

As far as program verification is concerned I present nothing essentially new. In testing a general purpose object (be it a piece of hardware, a program, a machine, or a system), one cannot subject it to all possible cases: for a computer this would imply that one feeds it with all possible programs! Therefore one must test it with a set of relevant test cases. What is, or is not, relevant cannot be decided as long as one regards the mechanism as a black box; in other words, the decision has to be based upon the internal structure of the mechanism to be tested. It seems to be the designer's responsibility to construct his mechanism in such a way—i.e. so effectively structured—that at each stage of the testing procedure the number of relevant test cases will be so small that he can try them all and that what is being tested will be so perspicuous that he will not have overlooked any situation. I have presented a survey of our system because I think it a nice example of the form that such a structure might take.

In my experience, I am sorry to say, industrial software makers tend to react to the system with mixed feelings. On the one hand, they are inclined to think that we have done a kind of model job; on the other hand, they express doubts whether the techniques used are applicable outside the sheltered atmosphere of a University and express the opinion that we were successful only because of the modest scope of the whole project. It is not my intention to underestimate the organizing ability needed to handle a much bigger job, with a lot more people, but I should like to venture the opinion that the larger the project, the more essential the structuring! A hierarchy of five logical levels might then very well turn out to be of modest depth, especially when one designs the system more consciously than we have done, with the aim that the software can be smoothly adapted to (perhaps drastic) configuration expansions.

Acknowledgments

I express my indebtedness to my five collaborators, C. Bron, A. N. Habermann, F. J. A. Hendriks, C. Ligtmans, and P. A. Voorhoeve. They have contributed to all stages of the design, and together we learned the art of reasoning needed. The construction and verification was entirely their effort; if my dreams have come true, it is due to their faith, their talents, and their persistent loyalty to the whole project.

Finally I should like to thank the members of the program committee, who asked for more information on the synchronizing primitives and some justification of my claim to be able to prove logical soundness a priori. In answer to this request an appendix has been added, which I hope will give the desired information and justification.

18.A Appendix

18.A.1 Synchronizing Primitives

Explicit mutual synchronization of parallel sequential processes is implemented via so-called "semaphores." They are special purpose integer variables allocated in the universe in which the processes are embedded; they are initialized (with the value 0 or 1) before the parallel processes themselves are started. After this initialization the parallel processes will access the semaphores only via two very specific operations, the so-called synchronizing primitives. For historical reasons they are called the *P*-operation and the *V*-operation.

A process, "*Q*" say, that performs the operation "*P* (sem)" decreases the value of the semaphore called "sem" by 1. If the resulting value of the semaphore concerned is nonnegative, process *Q* can continue with the execution of its next statement; if, however, the resulting value is negative, process *Q* is stopped and booked on a waiting list associated with the semaphore concerned. Until further notice (i.e. a *V*-operation on this very same semaphore), dynamic progress of process *Q* is not logically permissible and no processor will be allocated to it (see above "System Hierarchy," at level 0).

A process, "*R*" say, that performs the operation "*V* (sem)" increases the value of the semaphore called "sem" by 1. If the resulting value of the semaphore concerned is positive, the *V*-operation in question has no further effect; if, however, the resulting value of the semaphore concerned is nonpositive, one of the processes booked on its waiting list is removed from this waiting list, i.e. its dynamic progress is again logically permissible and in due time a processor will be allocated to it (again, see above "System Hierarchy," at level 0).

Corollary 1 *If a semaphore value is nonpositive its absolute value equals the number of processes booked on its waiting list.*

Corollary 2 *The P-operation represents the potential delay, the complementary V-operation represents the removal of a barrier.*

Note 1. *P*- and *V*-operations are “indivisible actions”; i.e. if they occur “simultaneously” in parallel processes they are noninterfering in the sense that they can be regarded as being performed one after the other.

Note 2. If the semaphore value resulting from a *V*-operation is negative, its waiting list originally contained more than one process. It is undefined—i.e. logically immaterial—which of the waiting processes is then removed from the waiting list.

Note 3. A consequence of the mechanisms described above is that a process whose dynamic progress is permissible can only lose this status by actually progressing, i.e. by performance of a *P*-operation on a semaphore with a value that is initially nonpositive.

During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.

18.A.2 Mutual Exclusion

In the following program we indicate two parallel, cyclic processes (between the brackets “**parbegin**” and “**parend**”) that come into action after the surrounding universe has been introduced and initialized.

```
begin semaphore mutex; mutex := 1;
  parbegin
    begin L1:   P (mutex); critical section 1;   V (mutex);
      remainder of cycle 1; go to L1
    end;
    begin L2:   P (mutex); critical section 2;   V (mutex);
      remainder of cycle 2; go to L2
    end
  parend
end
```

As a result of the *P*- and *V*-operations on “mutex” the actions, marked as “critical sections” exclude each other mutually in time; the scheme given allows straightforward extension to more than two parallel processes, the maximum value of mutex equals 1, the minimum value equals – $(n - 1)$ if we have n parallel processes.

Critical sections are used always, and only for the purpose of unambiguous inspection and modification of the state variables (allocated in the surrounding universe) that describe the current state of the system (as far as needed for the regulation of the harmonious cooperation between the various processes).

18.A.3 Private Semaphores

Each sequential process has associated with it a number of private semaphores and no other process will ever perform a *P*-operation on them. The universe initializes them with the value equal to 0, their maximum value equals 1, and their minimum value equals −1.

Whenever a process reaches a stage where the permission for dynamic progress depends on current values of state variables, it follows the pattern:

```
P (mutex);
“inspection and modification of state variables including a conditional
  V (private semaphore)”;
V (mutex);
P (private semaphore).
```

If the inspection learns that the process in question should continue, it performs the operation “*V* (private semaphore)”—the semaphore value then changes from 0 to 1—otherwise, this *V*-operation is skipped, leaving to the other processes the obligation to perform this *V*-operation at a suitable moment. The absence or presence of this obligation is reflected in the final values of the state variables upon leaving the critical section.

Whenever a process reaches a stage where as a result of its progress possibly one (or more) blocked processes should now get permission to continue, it follows the pattern:

```
P (mutex);
“modification and inspection of state variables including zero or more
  V-operations on private semaphores of other processes”;
V (mutex).
```

By the introduction of suitable state variables and appropriate programming of the critical sections any strategy assigning peripherals, buffer areas, etc. can be implemented.

The amount of coding and reasoning can be greatly reduced by the observation that in the two complementary critical sections sketched above the same inspection can be performed by the introduction of the notion of “an unstable situation," such as a free reader and a process needing a reader. Whenever an unstable situation emerges it is removed (including one or more V-operations on private semaphores) in the very same critical section in which it has been created.

18.A.4 Proving the Harmonious Cooperation

The sequential processes in the system can all be regarded as cyclic processes in which a certain neutral point can be marked, the so-called "homing position," in which all processes are when the system is at rest.

When a cyclic process leaves its homing position "it accepts a task"; when the task has been performed and not earlier, the process returns to its homing position. Each cyclic process has a specific task processing power (e.g. the execution of a user program or unbuffering a portion of printer output, etc.).

The harmonious cooperation is mainly proved in roughly three stages.

(1) It is proved that although a process performing a task may in so doing generate a finite number of tasks for other processes, a single initial task cannot give rise to an infinite number of task generations. The proof is simple as processes can only generate tasks for processes at lower levels of the hierarchy so that circularity is excluded. (If a process needing a segment from the drum has generated a task for the segment controller, special precautions have been taken to ensure that the segment asked for remains in core at least until the requesting process has effectively accessed the segment concerned. Without this precaution finite tasks could be forced to generate an infinite number of tasks for the segment controller, and the system could get stuck in an unproductive page flutter.)

(2) It is proved that it is impossible that all processes have returned to their homing position while somewhere in the system there is still pending a generated but unaccepted task. (This is proved via instability of the situation just described.)

(3) It is proved that after the acceptance of an initial task all processes eventually will be (again) in their homing position. Each process blocked in the course of task execution relies on the other processes for removal of the barrier. Essentially, the proof in question is a demonstration of the absence of "circular waits": process P waiting for process Q waiting for process R waiting for process P. (Our usual term for the circular wait is "the Deadly Embrace.") In a more general society than our system this proof turned out to be a proof by induction (on the level of hierarchy, starting at the lowest level), as A. N. Habermann has shown in his doctoral thesis.

19

Self-stabilizing Systems in Spite of Distributed Control

Edsger W. Dijkstra

Key Words and Phrases

Multiprocessing, networks, self-stabilization, synchronization, mutual exclusion, robustness, sharing, error recovery, distributed control, harmonious cooperation, self-repair

CR Categories

4.32

The synchronization task between loosely coupled cyclic sequential processes (as can be distinguished in, for instance, operating systems) can be viewed as keeping the relation "the system is in a legitimate state" invariant. As a result, each individual process step that could possibly cause violation of that relation has to be preceded by a test deciding whether the process in question is allowed to proceed or has to be delayed. The resulting design is readily—and quite systematically—implemented if the different processes can be granted mutually exclusive access to a common store in which "the current system state" is recorded.

A complication arises if there is no such commonly accessible store and, therefore, "the current system state" must be recorded in variables distributed over the various processes; and a further complication arises if the communication facilities are limited in the sense that each process can only exchange information

Originally published in Communications of the ACM, Volume 17 / Number 11 / November 1974.

with "its neighbors," i.e. a small subset of the total set of processes. The complication is that the behavior of a process can only be influenced by that part of the total current system state description that is available to it; local actions taken on account of local information must accomplish a global objective. Such systems (with what is quite aptly called "distributed control") have been designed, but all such designs I was familiar with were not "self-stabilizing" in the sense that, when once (erroneously) in an illegitimate state, they could—and usually did!—remain so forever. Whether the property of self-stabilization—for a more precise definition, see below—is interesting as a starting procedure, for the sake of robustness or merely as an intriguing problem, falls outside the scope of this article. It could be of relevance on a scale ranging from a worldwide network to common bus control. (I have been told that the first solution shown below was used a few weeks after its discovery in a system where two resource-sharing computers were coupled via a rather primitive channel along which they had to arrange their cooperation.)

We consider a connected graph in which the majority of the possible edges are missing and a finite state machine is placed at each node; machines placed in directly connected nodes are called each other's neighbors. For each machine one or more so-called "privileges" are defined, i.e. boolean functions of its own state and the states of its neighbors; when such a boolean function is true, we say that the privilege is "present." In order to model the undefined speed ratios of the various machines, we introduce a central daemon—its replacement by a distributed daemon falls outside the scope of this article—that can "select" one of the privileges present. The machine enjoying the selected privilege will then make its "move"; i.e. it is brought into a new state that is a function of its old state and the states of its neighbors. If for such a machine more than one privilege is present, the new state may also depend on the privilege selected. After completion of the move, the daemon will select a new privilege.

Furthermore there is a global criterion, telling whether the system as a whole is in a "legitimate" state. We require that: (1) in each legitimate state one or more privileges will be present; (2) in each legitimate state each possible move will bring the system again in a legitimate state; (3) each privilege must be present in at least one legitimate state; and (4) for any pair of legitimate states there exists a sequence of moves transferring the system from the one into the other.

We call the system "self-stabilizing" if and only if, regardless of the initial state and regardless of the privilege selected each time for the next move, at least one privilege will always be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves. For more than a year—at least to my knowledge—it has been an open question whether nontrivial (e.g. all states legitimate is considered trivial) self-stabilizing systems could exist. It is not directly obvious whether the local moves can assure convergence toward satisfaction of such a global criterion; the nondeterminacy as embodied by the daemon is an added complication. The question is settled by each of the following three constructs. For brevity's sake most of the heuristics that led me to find them, together with the proofs that they satisfy the requirements, have been omitted and—to quote Douglas T. Ross's comment on an earlier draft, "the appreciation is left as an exercise for the reader." (For the cyclic arrangement discussed below the discovery that not all machines could be identical was the crucial one.)

In all three solutions we consider $N + 1$ machines, numbered from 0 through N. In order to avoid avoidable subscripts, I shall use for machine $nr\ .\ i$:

L: to refer to the state of its lefthand neighbor, machine $nr\ .\ (i - 1)\mathbf{mod}(N + 1)$,

S: to refer to the state of itself, machine $nr\ .\ i$,

R: to refer to the state of its righthand neighbor, machine $nr\ .\ (i + 1)\mathbf{mod}(N + 1)$.

In other words, we confine ourselves to machines placed in a ring (a ring being roughly the sparsest connected graph I could think of); machine $nr\ .\ 0$ will also be called "the bottom machine," machine $nr\ .\ N$ will also be called "the top machine." For the legitimate states, I have chosen those states in which exactly one privilege is present. In describing the designs we shall use the format:

if *privilege* **then** *corresponding move* **fi**.

Solution with K-state Machines (K > N)

Here each machine state is represented by an integer value S, satisfying $0 \leq S < K$. For each machine, one privilege is defined, viz.
for the bottom machine:

if $L = S$ **then** $S := (S + 1)\mathbf{mod}\ K$ **fi**

for the other machines:

if $L \neq S$ **then** $S := L$ **fi**

Note 1. With a central daemon the relation $K \geq N$ is sufficient.

Note 2. This solution has been generalized by C.S. Scholten [1] for an arbitrary network in which the degree of freedom in the legitimate state is that of the special Petri-nets called "event graphs": along each independent cycle the number of privileges eventually converges toward an arbitrary predetermined constant.

Solution with Four-state Machines

Here each machine state is represented by two booleans xS and upS. For the bottom machine $upS =$ **true** by definition, for the top machine $upS =$ **false** by definition: these two machines are therefore only two-state machines. The privileges are defined as follows:
for the bottom machine:

```
    if $xS = xR$ and non $upR$ then $xS :=$ non $xS$ fi
```

for the top machine:

```
    if $xS \neq xL$ then $xS :=$ non $xS$ fi
```

for the other machines:

```
    if $xS \neq xL$ then $xS :=$ non $xS$; $upS :=$ true fi;
    if $xS = xR$ and $upS$ and non $upR$ then $upS :=$ false fi
```

The four-state machines may enjoy two privileges. The neighbor relation between bottom and top machines is not exploited; we may merge them into a single machine, which is then also a four-state machine for which also two privileges have been defined.

Solution with Three-state Machines

Here each machine state is represented by an integer value S, satisfying $0 \leq S < 3$. The privileges are defined as follows:
for the bottom machine:

```
    if $(S + 1)$mod $3 = R$ then $S := (S - 1)$mod $3$ fi
```

for the top machine:

```
    if $L = R$ and $(L + 1)$mod $3 \neq S$ then $S := (L + 1)$mod $3$ fi
```

for the other machines:

```
    if $(S + 1)$mod $3 = L$ then $S := L$ fi;
    if $(S + 1)$mod $3 = R$ then $S := R$ fi
```

Again the machine $nr \cdot i$ with $0 < i < N$ may enjoy two privileges: the neighbor relation between bottom and top machine has been exploited.

Acknowledgments are due to C.S. Scholten who unmasked an earlier effort as fallacious and since then has generalized the first solution, to C.A.R. Hoare and M. Woodger whose fascination by the first two solutions was an incentive to find

the third one, and to the referees whose comments regarding the presentation of these results have been most helpful.

Received November 1973; revised June 1974

Reference

3. Scholten, C.S. Private communication.

20 On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens

As an example of cooperation between sequential processes with very little mutual interference despite frequent manipulations of a large shared data space, a technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.

Key Words and Phrases

Multiprocessing, fine-grained interleaving, cooperation between sequential processes with minimized mutual exclusion, program correctness for multiprogramming tasks, garbage collection

CR Categories

4.32, 4.34, 4.35, 4.39, 5.24

20.1 Introduction

In any large-scale computer installation today, a considerable amount of time of the (general purpose) processor is spent on "operating the system." With the advent

Originally published in Communications of the ACM, Volume 21 / Number 11 / November 1978.

of multiprocessor installations, the question arises to what extent such "housekeeping activities" can be carried out concurrently with the computation(s) proper. One of the problems that have to be dealt with is organizing the cooperation of the concurrent processes so as to keep exclusion and synchronization constraints extremely weak, in spite of very frequent manipulations (by all processes involved) of a large shared data space. The problem of garbage collection was selected as one of the most challenging problems in this respect (and hopefully a very instructive one). Our exercise has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap possible. In our presentation we have tried to blend a condensed design history—so as not to hide the heuristics too much—with a rather detailed justification of our final solution. We have tried to keep exclusion and synchronization constraints between the processes as weak as possible, and how to deal with the complexities engendered by doing so is the main topic of this paper.

It has hardly been our purpose to contribute specifically to the art of garbage collection, and consequently no practical significance is claimed for our solution. For that reason we felt justified in tackling a specific form of the garbage collection problem as it presents itself in the traditional implementation environment of pure Lisp. We are aware of the fact that we have left out of consideration several aspects of the garbage collection problem that are important from other points of view (see, for instance, [4]).

In our abstract form of the problem, we consider a directed graph of varying structure but with a fixed number of nodes, in which each node has at most two outgoing edges. More precisely, each node may have a left-hand outgoing edge and may have a right-hand outgoing edge, but either of them or both may be missing. In this graph a fixed set of nodes exists, called "the roots." A node is called "reachable" if it is reachable from at least one root via a directed path along the edges. The subgraph consisting of all reachable nodes and their interconnections is called "the data structure;" nonreachable nodes, i.e. nodes that do not belong to the data structure, are called "garbage nodes." The data structure can be modified by actions of the following types:

(1) Redirecting an outgoing edge of a reachable node towards an already reachable one.

(2) Redirecting an outgoing edge of a reachable node towards a not yet reachable one without outgoing edges.

(3) Adding—where an outgoing edge was missing—an edge pointing from a reachable node towards an already reachable one.

(4) Adding—where an outgoing edge was missing—an edge pointing from a reachable node towards a not yet reachable one without outgoing edges.

(5) Removing an outgoing edge of a reachable node.

In actions (1), (2), and (5) nodes may be disconnected from the data structure and thus become garbage. In actions (2) and (4) a garbage node is "recycled," i.e. made reachable again.

The representation of the graph is such that each node can be identified independently of the structure of the graph, and that finding the left- or right-hand successor of a node can be regarded as a primitive operation, whereas finding its predecessor nodes would imply a search through the complete collection of nodes. Because of this representation, finding garbage is a nontrivial task, which is delegated to a so-called "garbage collector." The garbage collector maintains a so-called "free list," i.e. a collection of nodes that have been identified as garbage and are available to be added to the data structure.

In classical Lisp implementations the computation proper (i.e. the modifications of the data structure as described above) proceeds until the free list is exhausted (or nearly so). Then the computation proper comes to a grinding halt, after which garbage is collected: starting from the roots, all reachable nodes are marked; upon completion of this marking cycle all unmarked nodes can be concluded to be garbage, and are appended to the free list, after which the computation proper is resumed.

The minor disadvantage of this arrangement is the delay of the computation proper; its major disadvantage is the unpredictability of these garbage collecting interludes, which makes it hard to design such systems so as to meet real-time requirements as well. It was therefore tempting to investigate whether a second processor—called "the collector"—could collect garbage concurrently with the activity of the other processor—for the purpose of this discussion called "the mutator"—which would be dedicated to the computation proper. In order to investigate an exemplary problem, we have imposed upon our solution a number of constraints (compare [4]).

First, we wanted the synchronization and exclusion constraints between the mutator and the collector to be as weak as possible. (The classical implementation presents in this respect the other extreme: a garbage collecting interlude can in its entirety be regarded as a single critical section that excludes all mutator activity!) We wanted in particular to avoid highly frequent mutual exclusion of "elaborate" activities, as this would defy our aim of concurrent activity: our ultimate aim was something like no more interference than the mutual exclusion of a single read and/or write of the same single variable. One synchronization measure is evidently unavoidable: when needing a new node from the free list, the mutator may have to be delayed until the collector has appended some nodes to the free list.

Second, we wanted to keep the overhead on the activity of the mutator (as required for the cooperation with the collector) as small as possible.

Third, we did not want the mutator's ongoing activity to impair the collector's ability to identify garbage more than we could avoid. With a major cycle of the collector consisting of a marking phase followed by an appending phase, it is impossible to guarantee that the appending phase will append *all* garbage existing at its beginning: new garbage could have been created between an appending phase and the preceding marking phase. We do require, however, that such garbage, existing at the beginning of an appending phase but not identified as such by the collector, will be appended in the next major cycle of the collector. Moreover, we have rejected solutions in which garbage created during a marking phase was guaranteed *not* to be appended during the next appending phase.

20.2 The Grain of Action

The fact that we require concurrent operation of two or more processes raises the problem of defining the net effect of such concurrent operation. In order to explain the problem we introduce the terms "local variable" for those variables that are accessed by one process only, and "shared variable" for those that are accessed by at least two processes.

As long as our concurrent processes only operate on local variables, there is no problem: we suppose that no one will have any doubt as to the net result of the concurrent operation of the two programs S_0 and S_1 given by

$$S_0\colon x := 0 \quad \text{and} \quad S_1\colon y := 3$$

This, however, changes radically as soon as we consider shared variables. With shared "z" some readers may assume that the concurrent operation of S_2 and S_3, given by

$$S_2\colon z := 0 \quad \text{and} \quad S_3\colon z := 3$$

will yield either $z = 0$ or $z = 3$, but in that case we must destroy that illusion! We need only assume z to consist of two bits z_0 and z_1 ($z = 2z_1 + z_0$), and S_2 and S_3 on closer scrutiny to be composed as follows:

$$\begin{array}{lcl} S_2\colon z_0 := 0; & \quad\text{and}\quad & S_3\colon z_0 := 1; \\ z_1 := 0 & & z_1 := 1 \end{array}$$

to reach the conclusion that $z = 1$ and $z = 2$ are also possible results.

In order to express our intentions unambiguously, we introduce the notion of "atomic operations," denoted in this paper by a piece of program placed between a pair of angle brackets (we do not allow nested use of such pairs). We further require all accesses to shared variables to be part of an atomic operation and postulate that the net effect of our concurrently operating processes is as if atomic operations are mutually exclusive, i.e. the execution periods of atomic operations don't overlap. (We note in passing that it is pointless to introduce atomic operations accessing local variables only.) As a result it is now clear that concurrent operation of

$$S_2\colon \langle z := 0\rangle \quad \text{and} \quad S_3\colon \langle z := 3\rangle$$

will, indeed, yield either $z = 0$ or $z = 3$ (even if, upon closer scrutiny, the assignments to z turn out to be composed of successive operations on the individual bits).

Having introduced atomic operations, we are now in a position to define a (partial) ordering between programs based on the notions "coarser-grained" and "finer-grained" ("A is coarser-grained than B" is equivalent to "B is finer-grained than A"). We say that A is coarser-grained than B (or alternatively, "has a coarser grain of action") if B is the result of replacing an atomic operation of A by a piece of program containing at least two atomic operations, and having *all by itself* the same net effect as the original operation.

Since a possible sequencing of the atomic operations in a coarse-grained solution of a problem can always be regarded as a possible sequencing of the atomic operations in a finer-grained solution, the proof that the finer-grained solution is correct implies the same for the coarse-grained solution. Hence the advantage of coarser-grained solutions is that their correctness proofs are easier than those for finer-grained ones; their disadvantage, however, is that their implementation usually requires more severe mutual exclusion measures, which tend to defeat the aim of concurrency.

20.3 Reformulation of the Problem

Our first step was to restate the problem in as simple a form as we could. We found two important simplifications.

First, we followed the not unusual practice of introducing a special root node, called "*NIL*," whose two outgoing edges point to itself, and representing a formerly missing edge now by an edge with the node *NIL* as its target. (In order to shorten our discussions we use the terms "source" and "target" of an edge: if an edge points from node A to node B, then A is said to be the source and B is said to be the target of that edge.) For us, the introduction of the node *NIL* was definitely much more than just a coding trick. It allowed us to view data structure modifications of types (3) and (5) as special cases of type (1), and those of type (4) as special cases of type (2), so that we were left with only two types of modification. In the sequel it will become clear that the reduced diversity thus achieved has been absolutely essential for our purposes.

A second simplification was obtained by viewing the nodes of the free list no longer as garbage, but as part of the data structure. This was achieved by introducing one or more special root nodes, and by linking the free nodes in such a way that *NIL* and all free nodes, *but no others*, are reachable from these special root nodes. This implies that from now on the nodes on the free list are reachable, and thus considered to be part of the data structure. A modification of type (2) is now replaced by a sequence of modifications of type (1): first redirecting an edge towards a node in the free list, then redirecting edges of free list nodes so as to remove that node from the free list. (Note that the operations must be performed in such an order that the node in question remains permanently reachable.) Making the free list part of the data structure is again no mere coding trick. It allowed us to eliminate modifications of type (2): now only *one* type of modification of the data structure is left to the mutator, namely type (1) "redirecting an outgoing edge of a reachable node towards an already reachable one." (Even the actions of the collector, required for appending an identified garbage node to the free list, are very close to the one operation available to the mutator. The only difference is that we have to allow the collector to redirect the outgoing edge of a reachable node towards a not yet reachable one.)

The activities of mutator and collector can now be described as repeated executions of

mutator: "redirect an outgoing edge of a reachable node towards an already reachable one"

collector: marking phase:
"mark all reachable nodes";
appending phase:
"append all unmarked nodes to the free list and remove the markings from all marked nodes".

The mutator and the collector must cooperate in such a fashion that the following two correctness criteria are satisfied.

CC1: Every garbage node is eventually appended to the free list. More precisely, every garbage node present at the beginning of an appending phase will have been appended by the end of the next appending phase.

CC2: Appending a garbage node to the free list is the collector's only modification of (the shape of) the data structure.

Our final goal was a fine-grained solution in which each atomic operation would be something like a single read or write of a variable. More precisely, we wanted in our final solution to accept the following atomic operations: "redirecting an edge," "finding the (left- or right-hand) successor of a node," and "testing and/or setting certain attributes of a node." (The latter class of operations is obviously needed for marking the nodes.) The implementation of these atomic operations falls outside the scope of this paper.

Moreover, we allow ourselves the (not essential but convenient) luxury of considering "append node nr. *i* to the free list" to be an atomic operation available to the collector. We felt entitled to do so because its finer-grained implementation in terms of a succession of redirection of edges is simple provided the free list remains long enough, for then the nodes involved are not touched by the mutator. Nor do we describe how to prevent the free list from getting too short, i.e. how to delay the mutator, if necessary, when it is about to take a node from the free list. (The latter is the familiar consumer/producer coupling, a fine-grained solution of which has been given in [3].)

We have taken the course of first finding a coarse-grained solution, and then transforming it into a finer-grained one. This two-stage approach has been of great heuristic value; it was, however, not without pitfalls.

20.4 The First Coarse-Grained Solution

A counterexample taught us that the goal "no extra overhead for the mutator" is unattainable. Suppose that the nodes *A* and *B* are permanently reachable via a constant set of edges, while node *C* is initially reachable only via an edge from *A* to *C*. Suppose furthermore that, from then on, the mutator performs repeatedly a sequence of redirections with the following results:

(1) making an outgoing edge from *B* point to *C*

(2) making the edge from *A* to *C* disappear

(3) making an outgoing edge from *A* point to *C*

(4) making the edge from *B* to *C* disappear.

Since the collector observes nodes one at a time, it may never discover that *C* is reachable: while the collector is observing *A* for its successors, *C* may be reachable via *B* only, and the other way round. We therefore expect that the mutator may have to mark in some way the target nodes of edges it redirects.

Marking will be described in terms of colors. We start with all nodes white, and will design the algorithm so that the combined activity of the collector's marking phase and the mutator will make all reachable nodes black. All nodes that are still white after the marking phase will thus be garbage. For any repetitive process—and the marking phase certainly is one—we have always two concerns (see [1]): first, we must have a monotonicity argument on which to base our proof of termination, and second, we must find an invariant relation which is initially true and not destroyed during the repetition, so that it still holds upon termination. For the monotonicity argument we choose (fairly obviously) that during the marking phase no node will go back from black to white. Since we will soon introduce the color "gray," we restate this more generally as: "during the marking phase no node will become lighter." (Gray is darker than white and lighter than black.) For the invariant relation—which must be satisfied both before and after the collector's marking cycle—we must generalize the initial and final states of the marking cycle. Our first choice (perhaps less obvious, but not unnatural) was

P1: "no edge points from a black node to a white one."

Additional action is now required from the mutator when it is about to introduce an edge from a black node to a white one, since just placing it would cause a violation of P1. The monotonicity argument requires that the black source node of the new edge has to remain black, so P1 tells us that the target node of the new edge cannot be allowed to remain white. But the mutator cannot just make it black, because that could cause a violation of P1 between the new target node and its immediate successors. We therefore introduce the intermediate color "gray," and let the mutator change the new target's color from white to gray; for reasons of simplicity, the mutator shall do so independently of the color of the new edge's source. Our choice was a coarse-grained mutator that repeatedly performs the following atomic operation, in which "shading a node" means making it gray if it is white, and leaving it unchanged if it is gray or black:

M1: ⟨redirect an outgoing edge of a reachable node towards an already reachable one, and shade the new target⟩.

Note 1. Disregarding P1, the problem of node *C* from the counterexample at the beginning of this section could also have been solved by having the mutator shade the old target instead of the new one. This, however, would lead to a solution in which garbage created during a marking phase is guaranteed not to be collected during the next appending phase. Hence, we rejected this solution in accordance with the last sentence of Section 20.1. ■

We have decided that the collector's marking phase should make all reachable nodes black while keeping P1 invariant. This decision leads fairly directly to a coarse-grained collector. Like the mutator, the collector's marking phase uses the intermediate color gray to preserve P1. Gray nodes are then ones which must be made black, but which might still have white successors. Hence, whenever it encounters a gray node, the marking phase must make it black and shade its successors. For our coarse-grained collector, let the entire operation of blackening a gray node and shading its successors be a single atomic operation. Since the marking phase must make all gray nodes black, it can terminate only when there are no more gray nodes. The obvious way of trying to guarantee the absence of gray nodes is to let the marking phase terminate when the collector had observed all nodes in some order without finding any gray ones. This produces the collector given below; it is described with the "**if**...**fi**" and "**do**...**od**" constructs introduced in [1]. (The idea of "$B \longrightarrow S$," a so-called "guarded command," is that the statement list S is only eligible for execution in those initial states for which B is true. Below we need only a few simple cases. The repetitive construct "**do** $B \longrightarrow S$ **od**" is semantically equivalent to the now traditional "**while** B **do** S **od**." The alternative construct "**if** $B1 \longrightarrow S1 \,[]\, B2 \longrightarrow S2$ **fi**" requires $B1$ or $B2$ to hold to start with; if $B2$ were **non** $B1$, it would be semantically equivalent to the now traditional "**if** $B1$ **then** $S1$ **else** $S2$ **fi**.") As before, angle brackets are used to enclose atomic operations. Comments have been inserted between braces, and labels have been inserted for future reference.

The collector has two local integer variables i and k, and a local variable c of type color; the nodes are assumed to be numbered from 0 through $M - 1$. Our coarse-grained collector then repeatedly executes the following program:

```
marking phase:
begin {there are no black nodes}
  "shade all roots" {P1 and there are no white roots} ;
  i:= 0; k:= M;
marking cycle:
  do k > 0 → {P1 and there are no white roots}
    ⟨c:= color of node nr. i⟩;
    if c = gray → k:= M;
      C1: ⟨shade the successors of node nr. i and make node nr. i black⟩
    [] c ≠ gray → k := k − 1
    fi;
    i:= (i + 1) mod M
  od
end {P1 and there are no white roots and no gray nodes, hence—as is
  easily seen—all white nodes are garbage};
```

```
appending phase:
begin $i$:= 0;
  appending cycle:
    do $i < M \rightarrow$ {all nodes with a number $< i$ are nonblack; all nodes
        with a number $\geq i$ are nongray, and are garbage, if white}
      $\langle c$:= color of node nr. $i\rangle$;
      if $c$ = white $\rightarrow \langle$append node nr. $i$ to the free list$\rangle$
      $[\,]c$ = black $\rightarrow \langle$make node nr. $i$ white$\rangle$
      fi;
      $i$:= $i + 1$
    od {there are no black nodes}
end
```

Note 2. Appending node nr. i to the free list includes redirecting its outgoing edges so that no other nodes than *NIL* or free nodes can be reached from it (see our second simplification as described in Section 20.3). ■

Note 3. It does not matter in which order nodes are examined during the marking phase, so we could have written a more general algorithm that does not specify any fixed order. Such an algorithm would allow more efficient implementations of the marking phase, in which the collector maintains a list of gray or probably gray nodes. For the sake of simplicity, we have not done so. ■

We shall now demonstrate that the correctness criteria CC1 and CC2 are met.

Proof. In order to prove that CC2 is met, we observe that, because in the marking phase the collector does not change (the shape of) the data structure, it suffices to show that during the appending phase it appends only garbage nodes to the free list. Because node nr. i is appended after having been observed to be white, it suffices to show that the relation "a white node with a number $\geq i$ is garbage" (1) is an invariant of the appending cycle's repeatable action, (2) holds when the collector enters its appending cycle.

(1) We shall demonstrate the invariance first, and shall do so by first proving it for the appending cycle's repeatable action in isolation, and then showing that the mutator leaves that proof's assertions invariant.

Because in the appending cycle's repeatable action i is increased, the collector could only violate the relation by making a nongarbage node white or by making a (white) garbage node into nongarbage. By the alternative construct either violation is possible, but only with respect to node nr. i; we can still guarantee "a white node with a number $> i$ is garbage," from which it follows that the subsequent increase $i := i + 1$ restores the original relation.

Because i is a local variable of the collector, also the mutator could only violate the assertions either by making a nongarbage node white—which it does not, because M1 only shades—or by making a (white) garbage node into nongarbage—which it does not either, because M1 only redirects edges towards already reachable nodes and, hence, leaves garbage garbage. Therefore the mutator's actions do not invalidate the demonstration that the relation is an invariant of the appending cycle's repeatable action.

(2) To show next that the relation holds at the beginning of the appending cycle, we have to demonstrate (because $i = 0$) that the marking phase has established that "all white nodes are garbage," which shall be shown under the assumption that, at the beginning of the marking phase, there were no black nodes.

Because the absence of black nodes implies P1, and because M1 and C1 have been carefully designed so as to leave P1 invariant and not to introduce white roots, "P1 and there are no white roots" is clearly established before and kept invariant during the marking cycle. When furthermore all gray nodes have disappeared, our target state, in which all reachable nodes are black and all white nodes are garbage, has been reached.

The marking cycle terminates with a scan past all nodes, during which no gray nodes are encountered. If we had only the collector to consider, the conclusion that at the end of such a scan gray nodes are absent—and hence the target state has been reached—would be trivial. Due to the ongoing activity of the mutator—the shading activity of which can introduce gray nodes!—a more subtle argument, which now follows, is required.

First, we observe that the target state, characterized by the absence of gray nodes, is stable: the absence of white reachable nodes prevents the mutator from introducing gray ones, and the absence of gray nodes prevents the collector from doing so.

Second, we show that a collector scan past all nodes, during which no gray nodes are encountered, implies that the stable target state has already been reached at the beginning of that scan: because the mutator leaves gray nodes gray and the collector did not color any nodes during that scan, a gray node existing at its beginning would, in contradiction to the assumption, have been encountered during that scan. Hence we can conclude that upon termination of the marking phase all white nodes are indeed garbage.

Because the appending phase makes all black nodes white, and the mutator does not introduce black nodes, there are no black nodes at the end of the appending phase; this justifies the assumption made above that there would be no black nodes at the beginning of the marking phase. Thus we have completed the proof that starting the collector in the absence of black nodes ensures that CC2 is met.

To prove that CC1 is met, we must first show that the collector's two phases terminate properly.

Proper termination of the appending phase is obvious, except for one thing: node nr. i must be black or white, because the alternative construct does not cater for the case "c = gray." But we have already proved that at the end of the marking phase, there are no gray nodes and every white node is garbage. Since the mutator cannot shade a garbage node, and shading a black node has no effect, it is clear that every node is either black or white when it is examined during the appending phase.

Termination of the marking phase follows from the fact that the integral quantity $k + M*$ (the number of nonblack nodes)—which, by definition, is nonnegative—is left invariant by the activity of the mutator, and is decreased by at least one in each iteration of the marking cycle.

Consider now the situation at the beginning of an appending phase. At that moment, the nodes are partitioned into three sets:

— The set of reachable nodes (they are black)

— The set of white garbage nodes (during the first appending phase to come, they will be appended to the free list)

— The set of black garbage nodes (during the first appending phase to come, they will not be appended to the free list, but they will be made white).

Calling the last set the set of "*D*-nodes," we have to show that all *D*-nodes will be appended during the second appending phase to come.

We call an edge "leading into *D*" when its target is a *D*-node, but its source is not. Because *D*-nodes are garbage, we can state that at the beginning of the first appending phase, sources of edges leading into *D* are white garbage nodes.

Since the *D*-nodes are garbage, the mutator will not redirect edges so as to make them point to a *D*-node, and since they are black to start with, during the first appending phase the collector will not do so either. The collector, however, will append all white garbage nodes, which includes—see note 2—redirecting outgoing edges of the nodes appended, so that, as a result, we can state that at the end of the first appending phase

— all *D*-nodes are white garbage nodes

— there are no edges leading into *D*.

The absence of edges leading into *D* is an invariant for the subsequent marking phase: the mutator does not introduce them, because *D*-nodes are garbage, and the collector does not redirect edges during its marking phase. The continued absence of edges leading into *D*, plus the fact that all *D*-nodes are white garbage to start with, implies that the *D*-nodes remain white garbage nodes during the subsequent marking phase: because they are garbage, the mutator leaves them as they are, and because they are all white, the collector is prevented from shading them. (Shading the first *D*-node by the collector would require the existence of an edge pointing to it from a gray node; in view of the absence of edges leading into *D*, this gray node would have to be a *D*-node, which is impossible.) Consequently, at the end of the marking phase, all *D*-nodes are still white garbage and will be appended to the free list during the subsequent appending phase. Hence, CC1 is also met. ■

By keeping P1 invariant during the marking cycle, we obtained our coarse-grained solution. Encouraged by this success, we tried to keep P1 also invariant in a finer-grained solution—a solution in which the mutator's action M1 was split into two atomic operations: one for redirecting the edge and one for shading the new target. In order to keep P1 invariant, the mutator had to shade the future target first, and then redirect the edge towards the node just shaded. This finer-grained solution—although presented in a way sufficiently convincing to fool ourselves—contained the following bug, discovered by N. Stenning and M. Woodger [5].

Consider the following sequence of events:

1. Prior to introducing an edge from node *A* to node *B*, the mutator shades node *B* (and goes to sleep)
2. The collector goes through a complete marking phase, followed by an appending phase (node *B* is now white, i.e. the mutator's shading has been undone! We further note that there is no garbage)
3. The collector goes through part of the next marking phase (and then goes to sleep), during which it so happens that node *A* is made black and node *B* is left white
4. The mutator (wakes up and) introduces without making garbage the edge from *A* to *B* (P1 is now violated)
5. The mutator removes all other ingoing edges of *B*—the absence of garbage makes this possible—and goes to sleep again (node *B* is now only reachable via the edge from *A*)
6. The collector completes its marking phase (node B has remained white)
7. The collector goes through its appending phase, during which the reachable node *B* is erroneously appended to the free list.

This ill-fated effort convinced us that in the finer-grained solution we were heading for, total absence of an edge from a black node to a white one was a stronger relation than we could maintain. However, it still seemed reasonable to retain the notion of “gray” as “semi-marked,” more precisely, as representing an unfulfilled marking obligation. This meant that we could use the same collector. However, we had to find a different coarse-grained mutator that we could use as a stepping stone to our ultimate fine-grained solution.

20.5 A New Coarse-Grained Solution

For our new coarse-grained solution, we had to replace P1 by a weaker relation. (It was replaced by P2 and P3, defined below.) In our first solution, we had made essential use of the fact that during the marking cycle, the validity of P1 allowed us to conclude that the existence of a white reachable node implied the existence of a gray node. (It even implied the existence of a gray reachable node, but the reachability of the gray node was not essential.) For our new solution we needed a weaker relation P2, from which the same conclusion could be drawn. We defined a “propagation path” to be one consisting solely of edges with white targets, and starting from a gray node, and chose the following relation:

P2: “for each white reachable node, there exists a propagation path leading to it.”

Note 4. The gray node of the propagation path is not necessarily reachable. ■

Corollary 1 *If each root is gray or black, the absence of edges from black to white clearly implies that relation P2 is satisfied. In particular, P2 is true at the beginning of the marking cycle, because all roots have been shaded and there are no black nodes.*

In proving the correctness of our first solution, the invariance of P1 was only needed to prove that, during the marking cycle, the absence of gray nodes implies that all white nodes are garbage. We can clearly use P2 instead of P1 to draw the same conclusion. Therefore, to prove the correctness of our new solution, we need only prove that both the mutator and the marking collector leave P2 invariant. However, P2 turned out to be too weak a relation from which to conclude that its truth will not be destroyed. To keep P2 invariant, we had to restrict the existence of black-to-white edges by the following further relation—analogous to P1, but weaker—

P3: “only the last edge placed by the mutator may lead from a black node to a white one.”

Corollary 2 *In the absence of black nodes, P3 is trivially satisfied. Hence it holds at the beginning of the marking cycle.*

By Corollaries 1 and 2, P2 **and** P3 is true at the beginning of the marking cycle. To show that the marking cycle of our coarse-grained collector leaves P2 **and** P3 invariant, we must show that the atomic operation C1 cannot destroy its truth. Shading a node can cause neither P2 nor P3 to become false; shading the successors of a node implies that its outgoing edges are no longer part of any propagation path, so making that node black immediately afterwards does not make P2 false either. Moreover, since its successors have just been shaded, making the node black does not introduce a black-to-white edge, and, hence, cannot make P3 false either. Combining these results we conclude that in its marking cycle the collector leaves P2 **and** P3 invariant.

All that remains to be done now in order to construct a correct coarse-grained solution is to define a mutator operation that leaves P2 **and** P3 invariant. When the mutator redefines an outgoing edge of a black node, it may direct it towards a white node. This new black-to-white edge is the one permitted by P3. We must prevent, however, the previously redirected edge from also being a black-to-white edge, and we therefore consider for our coarse-grained mutator the following atomic operation:

M2: ⟨shade the target of the previously redirected edge, and redirect an outgoing edge of a reachable node towards a reachable node⟩.

Note 5. For the very first time that the mutator redirects an edge, we can assume that (for lack of a previously redirected edge) either the shading will be suppressed or else an arbitrary reachable node will be shaded. The choice does not matter for the sequel. ■

Action M2 has been carefully chosen in such a way that it leaves P3 invariant. We now prove that it leaves the stronger relation P2 **and** P3 invariant as well, thereby showing that the new mutator with our original collector gives a correct solution.

Proof. The action M2 cannot introduce new reachable nodes. Hence, every white node which is reachable after the operation had a propagation path leading to it before the operation. If the node whose successor is redefined is black, then its outgoing edge was *not* part of any propagation path, so the edges of the old propagation paths will be sufficient to provide the propagation paths needed to maintain P2. (We may not need all of them because of the shading operation, and because some white reachable nodes may have been made unreachable.) If the node whose successor is redefined was white or gray to start with, then the net result of action M2 will be a graph without edges from a black node to a white one: if one existed, then its target has now been shaded, and no new one has been introduced since the source of the new edge is not black. The roots must still be gray or black, so by Corollary 1, P2 still holds. ■

20.6 A Fine-Grained Solution

To complete our task, we now use the coarse-grained solution of Section 20.5 as a stepping stone to a fine-grained one. For our fine-grained mutator, M2 is split up into the following succession of atomic operations:

M2.1: ⟨shade the target of the previously redirected edge⟩;
M2.2: ⟨redirect an outgoing edge of a reachable node towards a reachable node⟩

In the collector, we break open C1 as the sequence of five atomic operations (*m*1 and *m*2 being local variables of the collector):

C1.1: ⟨*m*1:= number of the left-hand successor of node nr. *i*⟩;
C1.2: ⟨shade node nr. *m*1⟩;
C1.3: ⟨*m*2:= number of the right-hand successor of node nr. *i*⟩;
C1.4: ⟨shade node nr. *m*2⟩;
C1.5: ⟨make node nr. *i* black⟩.

We first observe that the collector's action of shading a node commutes with any number of mutator actions M2.1 and M2.2; without loss of generality we can, therefore, continue our discussion as if the four atomic operations C1.1 through C1.4 were replaced by a succession of the following two atomic operations:

C1.1a: ⟨shade the left-hand successor of node nr. *i*⟩;
C1.3a: ⟨shade the right-hand successor of node nr. *i*⟩.

Examining the proof for our coarse-grained solution, it is clear that in order to prove the correctness of this fine-grained one, it suffices to prove that P2 **and** P3 is still invariant during the (fine-grained) marking cycle. We shall prove the invariance of P2 **and** P3 by proving the invariance of a stronger relation.

For the purpose of its definition, we first introduce the notion of so-called "*C*-edges." Loosely formulated, *C*-edges are the edges, the sources of which have been detected as gray by the collector's marking cycle. More precisely, the set of *C*-edges is empty at the beginning of the marking cycle, and the actions C1.1a and C1.3a add to it the left-hand and right-hand outgoing edge of node nr. *i*, respectively. Note that the formulation has been chosen so as to make it clear that, from C1.1a onwards, being a *C*-edge is a property of the left-hand outgoing edge of node nr. *i*, independent of the node it points to. In particular, being a *C*-edge is invariant with respect to redirection of that edge by the mutator.

Note 6. We only define the set of *C*-edges for *our* benefit. The set is not explicitly updated, although the collector could easily do so. In the jargon, the term "ghost variable" is sometimes used for such an entity. ■

The strengthened versions of P2 and P3 can now be formulated as follows:

P2a: "every root is gray or black, and for each white reachable node there exists a propagation path leading to it, containing no *C*-edges"

P3a: "there exists at most one edge "*E*" satisfying
pr: "(*E* is a black-to-white edge) **or**
(*E* is a *C*-edge with a white target);"
the existence of such an *E* satisfying pr implies that the mutator is between action M2.2 and the subsequent action M2.1, and that *E* is identical with the edge most recently redirected by the mutator."

We now prove that P2a **and** P3a holds when the collector executes its marking cycle.

Proof. We first observe that P2a holds at the beginning of the marking cycle, thanks to Corollary 1 and the fact that the set of *C*-edges is then empty. As there are neither *C*-edges nor black nodes at the beginning of the marking cycle, there is no edge *E* satisfying pr, so at the beginning of the marking cycle P3a holds as well.

We further make the general remark that none of the operations M2.1, M2.2, C1.1a, C1.3a, and C1.5 introduces new white reachable nodes. Consequently, when proving the invariance of P2a under these operations, it suffices to show that the existence, before the operation, of a propagation path without *C*-edges, leading to a reachable node that is white before and after the operation, implies the existence afterwards of such a propagation path leading to that node.

The invariance of P2a **and** P3a with respect to the three shading operations M2.1, C1.1a, and C1.3a can be dealt with simultaneously. Propagation paths leading to a reachable node that is white both before and after the shading operation are either left intact or are shortened by it. If these propagation paths did not contain *C*-edges before the shading operation, then they won't do so afterwards. The shading operations of the collector do create new *C*-edges, but these are *C*-edges with gray or black targets, and, therefore, cannot belong to any propagation path. This proves the invariance of P2a. Relation P3a is invariant as well, because the collector's shading acts introduce neither a black-to-white edge, nor a *C*-edge with a white target, and operation M2.1 only removes the edge *E* if it did exist.

Action C1.5 leaves P2a invariant: because the outgoing edges of the gray node nr. *i* are *C*-edges, they don't belong to existing propagation paths without *C*-edges, and hence making that node black leaves the existence of such paths unaffected.

Action C1.5 also leaves P3a invariant, as it introduces no new solutions E of pr: it may introduce a black-to-white edge, but then that edge was already a C-edge with a white target.

Action M2.2 leaves P3a invariant because, if P3a held before M2.2, no edge E satisfying pr existed, and the redirection can create at most one such edge. We finally prove the invariance of P2a under M2.2. If the edge to be redirected is a C-edge or if its source is black, it does not belong to a propagation path without C-edges. Since, furthermore, M2.2 does not create C-edges, the existence of such paths remains in this case unaffected. In the other case—i.e. if the edge to be redirected is not a C-edge and has a white or gray source—the already established invariance of P3a implies after M2.2 the absence of black-to-white edges and the absence of C-edges with a white target. In view of Corollary 1, these absences imply for all white reachable nodes the existence of propagation paths without C-edges. ■

20.6.1 In Retrospect

It has been surprisingly hard to find the published solution and justification. It was only too easy to design what looked—sometimes even for weeks and to many people—like a perfectly valid solution, until the effort to prove it to be correct revealed a (sometimes deep) bug. The reasoning we have used contains most of the ideas needed for a formal proof in the style of [3] or [2]. Because the inclusion of such a proof would result in a paper, possibly tedious, but in any case very different from what we intended to write this time, we have confined ourselves to our informal justification (which we do *not* regard as an adequate substitute for a formal correctness proof). Whether our stepwise approach is more generally applicable, is at the moment of writing still an open question.

When it is objected that we still needed rather subtle arguments, we can only agree wholeheartedly: all of us would have preferred a simpler argument! Perhaps we should conclude that constructions that give rise to such tricky problems are not to be recommended. One firm conclusion, however, can be drawn: to believe that such solutions can be found without a very careful justification is optimism on the verge of foolishness.

History and Acknowledgments. (As in this combination, this is our first exercise in international and intercompany cooperation, some internal credit is given as well.) After careful consideration of a wider class of problems the third and the fifth authors selected and formulated this problem, and did most of the preliminary investigations; the first author found a first solution during a discussion with the fifth author, W.H.J. Feijen, and M. Rem. It was independently improved by the second author—to give the free list a root and mark its nodes as well, was his suggestion—and, on a suggestion made by John M. Mazola, by the first and the third author. The first and the fourth merged these embellishments, but introduced the bug that was found by N. Stenning and M. Woodger. The final version and its justification are the result of several trans-Atlantic iterations. The active and inspiring interest shown by David Gries is mentioned in gratitude. As with each new version of the manuscript the proofs became simpler, we also express our indebtedness to R. Stockton Gaines, whose comments on an earlier version caused two further iterations.

20.6.2 Glossary of Names

Correctness Criteria

CC1: Every garbage node is eventually appended to the free list. More precisely, every garbage node present at the beginning of an appending phase will have been appended by the end of the next appending phase.

CC2: Appending a garbage node to the free list is the collector's only modification of (the shape of) the data structure.

Atomic Operations of the Mutator

M1: ⟨redirect an outgoing edge of a reachable node towards an already reachable one, and shade the new target⟩.

M2: ⟨shade the target of the previously redirected edge, and redirect an outgoing edge of a reachable node towards a reachable node⟩.

M2.1: ⟨shade the target of the previously redirected edge⟩.

M2.2: ⟨redirect an outgoing edge of a reachable node towards a reachable node⟩.

Atomic Operations of the Collector

C1: ⟨shade the successors of node nr. i and make node nr. i black⟩.

C1.1: ⟨$m1$:= number of the left-hand successor of node nr. i⟩.

C1.2: ⟨shade node nr. $m1$⟩.

C1.3: ⟨$m2$:= number of the right-hand successor of node nr. i⟩.

C1.4: ⟨shade node nr. $m2$⟩.

C1.5: ⟨make node nr. i black⟩.

C1.1a: ⟨shade the left-hand successor of node nr. i⟩.

C1.3a: ⟨shade the right-hand successor of node nr. i⟩.

Invariant Relations

P1: No edge points from a black node to a white one.

P2: For each white reachable node, there exists a propagation path leading to it.

P3: Only the last edge placed by the mutator may lead from a black node to a white one.

P2a: Every root is gray or black, and for each white reachable node there exists a propagation path leading to it, containing no *C*-edges.

P3a: There exists at most one edge "*E*" satisfying
pr: "(*E* is a black-to-white edge) **or**
(*E* is a *C*-edge with a white target)";
the existence of such an *E* satisfying pr implies that the mutator is between action M2.2 and the subsequent action M2.1, and that *E* is identical with the edge most recently redirected by the mutator.

Received July 1977; revised February 1978

References

1. Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM 18*, 8 (Aug. 1975), 453–457.
2. Gries, D. An exercise in proving parallel programs correct. *Comm. ACM. 20*, 12 (Dec. 1977), 921–930.
3. Lamport, L. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng. SE-3*, 2 (March 1977), 125–143.
4. Steele, Jr., G.L. Multiprocessing compactifying garbage collection. *Comm. ACM 18*, 9 (Sep. 1975), 495–508.
5. Woodger, M. Private communications.

21

On the Reliability of Programs

E. W. Dijkstra

All speakers of the lecture series have received very strict instructions as how to arrange their speech; as a result I expect all speeches to be similar to each other. Mine will not differ, I adhere to the instructions. They told us: first tell what you are going to say, then say it and finally summarize what you have said.

My story consists of four points.

1. I shall argue that our programs should be correct
2. I shall argue that debugging is an inadequate means for achieving that goal and that we must prove the correctness of programs
3. I shall argue that we must tailor our programs to the proof requirements
4. I shall argue that programming will become more and more an activity of mathematical nature.

The starting point of my considerations is to be found at the “software failure”. A few years ago the wide-spread existence of this regrettable phenomenon was established beyond doubt; as far as my information tells me, the software failure is still there as vigorous as ever and its effects are sufficiently alarming to justify our concern and attention. What, however, is it?

Depending on the specific instance of failure one chooses, it can be described in many ways. One of the most common forms starts with an exciting project, but as it proceeds, deadlines are violated and what started as a fascinating thriller

This is an unpublished paper Edsger W. Dijkstra, On the reliability of programs, EWD303, 1970. Available at https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF.
Republished with permission.
This chapter was produced using the transcription https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html of the original text produced by David J. Bentley and was checked against the original text. In the process minor typos were taken care of.

slowly turns into a drama, to be played by an ever increasing number of actors, the majority of which know perhaps their own part but have certainly lost their grasp on the meaning of the performance as a whole. At last the curtain falls, only because it is too late, but not because anything has really been completed, for the final piece of software is still full of bugs and will remain so for the rest of its days. There are other forms, but they all have in common, that it turns out to be very, very difficult to get the whole program working with an acceptable degree of reliability.

When we try to explain the present as the natural outcome of the recent past, we get a better understanding of what has happened. In the past ten, fifteen years, the power of commonly available computers has increased by a factor of a thousand. The ambition of society to apply these wonderful pieces of equipment has grown in proportion and the poor programmer, with his duties in this field of tension between equipment and goals, finds his task exploded in size, scope and sophistication. And the poor programmer just has not caught up. Looking backwards we must conclude that the difficulty of the tasks ahead have been grossly underestimated in the past. Extrapolations concerning the number and power of computers to be installed have been made, but society's preparation for this oncoming wave of machinery has been the call for more and more programmers, rather than for more capable ones, who derive their greater capability from a better understanding of the nature of the programming task.

It is alarming to see how little the average programmer's attitude towards his work has changed. The reason is twofold: many programmers have been attracted towards the profession a long time ago, and among them many did not have the intellectual growth potential needed to keep up with the changing profession. Besides that I am afraid that in many organizations young people are attracted as programmers, who are selected on account of obsolete aptitude tests; I still often hear that a successful programmer should be "puzzle-minded" whereas I have the feeling that a clear and systematic mind is more essential. A modern, competent programmer should not be puzzle-minded, he should not revel in tricks, he should be humble and avoid clever solutions like the plague. In the period under discussion higher level programming languages have been accepted as general programming tools and they have been hailed as a tremendous step forward. OK, but is it sufficient? Higher level programming languages enable us perhaps to cope with a factor of 10 in scope but not with a factor of 1000. In the old days, programs of more than one or two thousand assembly code instructions were horrors, but in the mean time higher level programmers produce – admittedly – large programs of exactly the same degree of unreadability and unreliability, in which the role of old machine-code tricks has been taken over by cunning higher level language tricks. Truly, I cannot see the difference... The conclusion is that, in spite of the factor thousand in scope, present day programming tries to solve its problems with the same old methods. And therefore, if we want to improve matters, we should make it our serious business to minimize the usage of what is now by far our scarcest resource, viz. brainpower. The burning question is "Can we get a better understanding of the nature of the programming task, so that by virtue of this understanding, programming becomes an order of magnitude easier, so that our ability to compose reliable programs is increased by a similar order of magnitude?".

The fact that program reliability becomes a key issue is not only shown by the evidence around us, it is also quite easy to see why. A very large program is, by necessity, composed of a large number, say N, individual components and the fact that the N is large implies that the individual program components must be produced with a very high confidence level. If for each individual component the probability of being right equals p, for the whole program the probability P of being right will satisfy

$$P \leq p^N \tag{21.1}$$

and if we want P to differ appreciably from zero, p must be very close to one, because N is so large. And we shall never be able to exploit the power of computers unless we can cope with the case of very large N.

A common approach to get a program correct is called "debugging" and when the most patent bugs have been found and removed one tries to raise the confidence level further by subjecting the program to numerous test cases. From the failures around us we can derive ample evidence that this approach is inadequate. To remedy the situation it has been suggested that what we really need are "automatic test case generators" by which the pieces of program to be validated can be exercised still more extensively. But will this really help? I don't think so.

When faced with a mechanism—be it hardware or software—one can ask oneself "How can I convince myself of its being correct?" As long as we regard the mechanism as a black box, the only thing we can do is subject it to all possible inputs and check whether it produces the correct outputs. But for the kind of mechanisms we are considering this is absolutely out of the question. I have a pet example to demonstrate this. At my University we have a machine and one should for instance like to know, whether the fixed point multiplication instruction works properly. The machine has a rather short word length of 27 bits, as a result there are only 2^{54} different fixed point multiplications possible. So, why not try them all? With 2^{14} multiplications per second, 2^{54} multiplications $= 2^{40}$ sec $= 10^{12}$ sec. $= 10^7$ days $=$ 30.000 years! It takes 30.000 years to have all possible multiplications performed just once. One of the consequences of this number is that in the whole life time of the machine, the number of fixed point multiplications actually performed by our machine is a truly negligible fraction of the set of possible multiplications. From a simple-minded point of view we are only interested in the correct execution of the tiny set of multiplication the machine is actually called to perform. But because in programming we think not in terms of numerical values but in terms of variables, we have abstracted from the values actually processed by the arithmetic unit and we are only allowed to make this abstraction when the multiplier would do *any* multiplication correctly. I make this point because it is often not realised that the at first sight extreme and ridiculous reliability requirements imposed by us on the hardware is a direct consequence of the fact that without it we could not afford this vital abstraction. Another consequence of the number of 30.000 years that sampling testing is hopelessly inadequate to convince ourselves of the correctness even of a simple piece of equipment as a multiplier: whole classes of in some sense critical cases can and will be missed! All this applies a fortiori to programs that claim to cope with many more cases and take more time for each single execution. The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence.

But as long as we regard the mechanism as a black box, testing is the only thing we can do. The conclusion is that we cannot afford to regard the mechanism as a black box, i.e. we have to take its internal structure into account. One studies its internal structure and on account of this analysis one convinces onself that if such and such cases work “all others must work as well.” That is, the internal structure is exploited to reduce the number of still necessary testcases, for all the other ones (the vast majority) one tries to convince onself by reasoning, the only problem being that the amount of reasoning often becomes excessive, with the sad result that bugs remain.

This function of the mechanism’s internal structure opens a new way to attack the reliability problem. Once we have seen that the confidence level can only be reached by virtue of the structure of the mechanism, that the extent to which the program correctness is not purely a function of its external specifications and behaviour, but depends critically upon its internal structure, then we can invert the question and ask ourselves “What forms of program structuring can we find, what elements of programming style and what forms of discipline, all for the benefit of the confidence level of the final product?”.

Instead of trying to devise methods to establish the correctness of arbitrary, given programs, we are now looking for the subclass of what I would like to call “intellectually manageable programs”, which can be understood and for which we can justify our belief in their proper operation under all circumstances without excessive amounts of reasoning. This is done in order to reduce the number of testcases needed; in the case of software I see no reason at all why this approach could not be so effective that the number of testcases is eventually reduced to zero, i.e. that correctness can be shown a priori. Already now, debugging strikes me as putting the cart before the horse: instead of looking for more elaborate debugging aids, I would rather try to identify and remove the more productive bug-generators!

In short, I suggest that the programmer should continue to understand what he is doing, that his growing product remains firmly within his intellectual grip. It is my sad experience that this suggestion is repulsive to the average experienced programmer, who clearly derives a major part of his professional excitement from *not* quite understanding what he is doing. In this streamlined age, one of our most undernourished psychological needs is the craving for Black Magic and apparently the automatic computer can satisfy this need for the professional software engineer, who is secretly enthralled by the gigantic risks he takes in his daring irresponsibility. For his frustrations I have no remedy......

We return to our question "Can the programmer arrange his activity in such a way that his growing product remains firmly in his intellectual grip, that he continues to understand what he is doing?" Well, let me state my firm belief: yes, he can, it is possible to increase our programming ability by an order of magnitude. And as a corrolary: there is no other way, for when the programmer loses his intellectual grip on his product, he will never get the program in such a state that we can rely upon it. I will now try to convey the quintessence of the considerations upon which this confidence is founded. The reliable design of a highly sophisticated program is anyway a difficult task and we could place our question in a much broader perspective and ask ourselves "How does the human mind invent something very intricate, how does the human mind think difficult thoughts?" Fascinating as those general questions are, I shall not touch upon them. I shall restrict myself to the more limited field of programming, because already in the programming field we can distinguish five elements of mental discipline, each of which is a great help in keeping our programs understandable. In order to be able to talk about them, I must name them; I have called them sequencing discipline, operational abstraction, representational abstraction, configurational abstraction and textual encapsulation. In order to avoid false hopes: all these elements are known, all higher level programming languages cater more or less successfully for some of them. It is just that by making these elements of discipline more explicit, that we can exploit them more consciously and that we can get more of a yardstick along which to compare the respective qualities of different programming languages.

The sequencing discipline first. We should be very well aware of the fact that although the written program is the final product that leaves the programmer's hand, the true subject matter of his trade is formed by the possible computations—the making of which he leaves to the machine—that may be evoked by his program.

Whenever we make a statement about the correctness of a program it is a statement about the corresponding computations that may be evoked by it. The subject matter is dynamic, it is the happening in time, while the last thing we can lay our hands on is the static program text. It is on account of the latter that we must be able to make assertions about the former. It is therefore essential to bridge the conceptual gap between the static program text and the dynamic computations evolving in time as effectively as possible. In order to do this I cannot recommend too heartily to abolish the **goto** statement and to restrict oneself for sequencing purposes to the conventional conditional clauses, alternative clauses, repetitional clauses and recursion. The **goto** statement has been identified as one of the combinatorial bug generators we have been looking for. I draw attention to the fact that recursion is grouped under the class "sequencing discipline", I will return to this in a moment when we have mentioned operational abstraction.

Operational abstraction is embodied in most programming languages by the subroutine mechanism. It is an explicit recognition of the fact that a total net effect can be effectuated as the cumulative effect of a sequence of subactions and that this can be done by virtue of what these subactions do for us, as distinct from how these subactions work. In a main program calling subroutines we have a level of discourse in which we can—and should—regard the subroutines as available primitives whose net effect is known and it is in terms of this knowledge that we can and must understand the main program. At that level of abstraction it should be of no concern how the subroutines work, that is only relevant on another, more detailed semantic level. The mixing of two such levels is one of the most common sources of program bugs. For the sake of completeness I mention that operational abstraction can also be recognized when the sequencing clauses are used, viz. in the relation between the whole construction and the statement controlled by the clause; in the latter case it is customary to represent the difference of level by indentation. And it is now clear, why recursion is treated as a sequencing discipline: in the definition of a recursive operator the level which describes how the operator works coincides with a level in which the operator is used and as a result we cannot distinguish the two different semantic levels.

Operational abstraction deals with the actions occurring in the computation and we have the two sides of the coin "What they do" versus "How they work". With respect to information stored inside the machine we have a similar coin with two sides. The only thing a computer can do for us is to manipulate but the only reason for manipulating symbols is that they stand for something else. Conceptually the program manipulates rather abstract objects, while in fact it does so in terms of a particular representation for them. It is a very common occurrence that alternative programs for the same job can be viewed as different refinements of the same algorithm, the same as long as the algorithm is expressed in terms of rather abstract values, only differing in the particular representations chosen to distinguish between these values. This is called representational abstraction; by virtue of it it is often possible that alternative programs for the same job—or as the case may be: programs for similar jobs—can share a large part of their correctness proofs. I have the feeling that representational abstraction is one of the most powerful techniques for understanding programs. Also, it seems a bit neglected. I do not know of any programming language—although it may exist—that caters for it in a convincing way. Records as structured data types do not answer its needs. With record we can introduce composite data types, OK, but the record structure chosen permeates the program using them. Also, much attention has been paid to the problem of proving formally the correctness of a program, but as far as I have seen these formal proofs, they were always tied down to the particular representation chosen in this program, because the formal proof did operate in terms of the primitive data elements. As a result changing the representation would require a completely new formal proof. It is this observation that may indicate the limits of applicability of highly formalized proof techniques. I have now mentioned two forms of abstraction; the inverse process, viz. choosing an algorithm for an action or choosing a representation for a variable is called "refinement", and so we have operational refinement and representational refinement. We should mention that representational refinement is always accompanied by operational refinement. On the abstract level we have the unanalysed variables with actions operating upon them: when a particular representation is chosen, our original actions have to be translated in terms of algorithms operating on the components of the chosen representation.

The fourth is configurational abstraction. To a given machine we can add a standard library of subroutines. If we do so, we have in a sense rebuilt our machine, we have extended its instruction repertoire. But layers of standard software can rebuild a given machine so much more drastically that a separate name seems appropriate and I have chosen "configurational abstraction" for that purpose. I am referring to the functions of an operating system that can rebuild a single processor installation into a multiple processor installation, that can rebuild a hierarchy of storage levels into a virtual store etc. Ideally, this rebuilding of a given hardware configuration into a virtual machine serves a double purpose. Firstly it enables us to map different configurations into the same virtual machine, thereby providing a step towards program portability; secondly the virtual machine should be much more attractive to use than the original machines. To my great regret I get the impression that the second requirement is often overlooked, at least not met.

Finally we have textual encapsulation. If we consider any program of a sizeable complexity, this program should not be regarded as an object all by itself but as a member of a class of neighbouring programs, alternative programs for either the same or similar jobs. It is becoming recognized more and more that it is highly desirable that the transformation of a program into another one of the family should only affect well isolated portions of the program text and should not require adjustments scattered all through the program text. This is one of the main aspects of modularity. For some program changes this goal is very hard to achieve but the objective is so clear now that I expect it to become easier in the future. We may either invent programming languages more suited to this flexibility requirement or we shall devise more elaborate means of exploiting automatic equipment for the composition of program texts.

With the above I touched on five, what I called "elements of a programming discipline", all of them ways by which we can increase the understandability of programs. They have to do with the ease with which we can understand what a program is doing, they have to do with the feasibility of correctness proofs. It is to the pattern of such proofs that we now turn our attention.

Let me first relate them to patterns of proof for single sequential programs, because the proving techniques for sequential programs seem to be better understood than those for a bunch of parallel programs. If we have a program consisting of a sequence of statements to be executed in the order in which they are given the proving technique is fairly well understood. If the net effect of the execution of the constituent statements is given, the net effect of their successive execution can be established straightforwardly by what I have called "Enumerative reasoning", our main concern being that the amount of reasoning needed for each step does not get excessive. It is my impression that the amount of reasoning needed can be reduced when we can group subsequences into compound statements, the net effect of which allows a compact formulation. This is generally true: any sizeable piece of program, or even a complete program package, is only a useful tool that can be used in a reliable fashion, provided that the documentation pertinent for the user is much shorter than the program text. If any machine or system requires a very thick manual, its usefulness becomes for that very circumstance subject to doubt!

Returning to the correctness proof: the compact formulation of the net effect of compound statements often requires the introduction of a new terminology and it is not unusual that this terminology is provided for by the mechanism of representational abstraction. So much for a piece of program text without any sequencing clauses. In the case of conditional clauses, alternative clauses, case constructions and repetitive clauses, the whole construction should be repeated as a single compound and its net effect should be formulated in such a way that it is no longer transparent that, internally, the construction is controlled by such a clause. In particular: if it is controlled by an alternative clause, the description of its net effect should be applicable regardless of which of the two paths is taken, if it is controlled by a repetitive clause, the description of its net effect should be applicable to all possible numbers of repetitions. This is because in the context in which these constructions occur, they occur on account of what they do for us and not on account of how they work. This is an application of operational abstraction possible thanks to our strict sequencing disciplines. We should make this operational abstraction very explicitly and very consciously: without it the number of cases to be distinguished between in our reasoning has a tendency to grow exponentially with the program text.

The conditional, the alternative and the case construction themselves can be understood by enumerative reasoning, the construction with the repetitive clause, however, requires a special pattern of reasoning for its understanding because it must cover all possible numbers of repetitions. Basically, mathematical induction is our only mental tool adequate for the purpose—and in the case of understanding a recursive subroutine mathematical induction is often applied explicitly—but in the case of loops the understanding can be speeded up by resorting to one or two theorems, thereby avoiding an explicit appeal to mathematical induction. They are invariance theorems. If it can be shown that a single execution of the repeatable statement will leave a relation between the values of some variables invariant—and this itself can be established by enumerative reasoning—and if furthermore it can be shown that this same relation is valid before entering the loop, then the said relation will also hold after termination of the repetition. This turns out to be a very powerful theorem.

I have mentioned the exploitation of invariant relations explicitly because it appears to be one of our more powerful tools when we wish to make assertions about the harmonious co-operation of a bunch of parallel programs. In a number of cases the harmonious co-operation between a number of parallel programs could be established by showing the invariance of a relation. In this relation two sorts of quantities occur, on the one hand variables, common to the programs, and on the other hand variables that are a function of the state of progress of the individual processes. The analysis of the requirements of such a proof, again, is very illuminating. Suppose that we want to establish the invariance of such a relation. As long as the variables occurring in the relation remain untouched, their values remain unchanged and the validity of the relation remains invariant. We therefore focus our attention upon all operators occurring in the totality of the bunch of parallel programs, that may influence the value of one or more of the variables concerned. Let their number be M. If all these operators leave the relation invariant, the relation will be invariant. This requires a study of M cases. But be careful! We are studying the net effect of a bunch of *parallel* programs, so in principle it is insufficient to establish the invariance of the relation under the execution of all possible combinations, i.e. 2^M (-1) cases. Here again we are faced with exponential growth of the number of cases to be distinguished between in our reasoning. As a result it makes all the difference in the world, whether one has primitives at one's disposal by means of which mutual exclusion in time can be guaranteed. And I am not sure whether all designers of multiprocessor machines, of programming languages for process control etc. have paid enough attention to this observation.

Our knowledge of how to prove the correctness of programs is far from complete but it is growing. It will have an increasing influence on the programs that are going to be produced. It is irrealistic, however, to expect all potential benefits from this approach before quite a lot of cleaning up of programming languages has been established. That such forms of cleaning up are perfectly possible has been shown by professor Niklaus Wirth from Zurich who has designed the programming language PASCAL. I quote from his introduction "The desire for a new language for the purpose of teaching programming is due to my deep dissatisfaction with the presently used major languages whose features and constructs too often cannot be explained logically and convincingly and which too often represent an insult to minds trained in systematic reasoning. Along with this dissatisfaction goes my conviction that the language in which the student is taught to express his ideas profoundly influences his habits of thought and invention, and that the disorder governing these languages directly imposes itself onto the programming style of the students. I am inclined to think that the lack of discipline and structure in professional programming style is the major reason for the appalling lack of reliability of practically all larger software products." End of quotation. We shall never learn to write good and convincing programs in FORTRAN—a programming tool which, indeed, was a great step forward when it was conceived some fifteen years ago but which, by now, should be regarded as a lower level language, as a low grade coding device. This programming tool, and the thinking habits induced by it, have grown hopelessly inadequate. As a teacher it is my job to help programmers in clearing up their own thinking. Often this is a highly rewarding activity, equally delightful and instructive for both parties. But when talking to the produce as grown in what is getting known as "the pure FORTRAN environment", I am usually baffled, for unsuspected depths of misunderstanding open themselves before my very eyes. It is well known that we don't gain automatically from experience, on the contrary, that the wrong experience may easily corrupt the soundness of our judgement. In the case of FORTRAN, it is my impression that its intellectually degrading influence is not commonly recognized, that too few people realize that the sooner we can forget that it ever existed, the better, as it is now too inadequate, too difficult and therefore too expensive and too risky to use.

Ladies and gentlemen, let me come to my final conclusions. Automatic computers are with us for twenty years and in that period of time they have proved to be extremely flexible and powerful tools, the usage of which seems to be changing the face of the earth (and the moon, for that matter!) In spite of their tremendous influence on nearly every activity whenever they are called to assist, it is my considered opinion that we underestimate the computer's significance for our culture as long as we only view them in their capacity of tools that can be used. In the long run that may turn out to be a mere ripple on the surface of our culture. They have taught us much more: they have taught us that programming any non-trivial performance is really very difficult and I expect a much more profound influence from the advent of the automatic computer in its capacity of a formidable intellectual challenge which is unequalled in the history of mankind. This opinion is meant as a very practical remark, for it means that unless the scope of this challenge is realized, unless we admit that the tasks ahead are so difficult that even the best of tools and methods will be hardly sufficient, the software failure will remain with us. We may continue to think that programming is not essentially difficult, that it can be done by accurate morons, provided you have enough of them, but then we continue to fool ourselves and no one can do so for a long time unpunished.

Finally, let me summarize as instructed. Reliability concerns force us to restrict ourselves to intellectually manageable programs. This faces us with the questions "But how do we manage complex structure intellectually? What mental aids do we have, what patterns of thought are efficient? What are the intrinsic limitations of the human mind that we had better respect?" Without knowledge and experience, such questions would be very hard to answer, but luckily enough, our culture harbours with a tradition of centuries an intellectual discipline whose main purpose it is to apply efficient structuring to otherwise intellectually unmanageable complexity. This discipline is called "Mathematics". If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is, indeed, the most effective way to come to grips with complexity, we have no choice any longer: we should reshape our field of programming in such a way that their methods of understanding become equally applicable, for there are no other means.

Thank you.

PART IV

BIOGRAPHICAL ESSAYS

22

Edsger Dijkstra, The Man Who Carried Computer Science on His Shoulders

Krzysztof R. Apt

As it turned out, the train I had taken from Nijmegen to Eindhoven arrived late. To make matters worse, I was then unable to find the right office in the university building. When I eventually arrived for my appointment, I was more than half an hour behind schedule. The professor completely ignored my profuse apologies and proceeded to take a full hour for the meeting. It was the first time I met Edsger Wybe Dijkstra.

At the time of our meeting in 1975, Dijkstra was 45 years old. The most prestigious award in computer science, the ACM Turing Award, had been conferred on him three years earlier. Almost twenty years his junior, I knew very little about the field —I had only learned what a flowchart was a couple of weeks earlier. I was a postdoc newly arrived from communist Poland with a background in mathematical logic and a plan to stay in the West. I left the meeting with two book recommendations and a copy of a recent research article Dijkstra had written. He also suggested that I learn the programming language Pascal.

Dijkstra passed away in 2002. During the 1970s and 1980s, at the height of his career, he was probably the most discussed scientist in his field. He was a pioneer and a genius whose work and ideas shaped the emerging field of computer science

Reprinted with kind permission from the Inference, https://inference-review.com/, where it appeared online on 28 September 2020 as https://inference-review.com/article/the-man-who-carried-computer-science-on-his-shoulders, as part of Vol. 5, No. 3, September 2020. In the original version it was incorrectly stated that the International Marktoberdorf Summer School were annual.

like few others. It was over the course of his career that computer science became a respectable and established discipline.

There is an enduring tension between the engineering view of computer science, which is focused on building software systems and hardware components, and the mathematical and logical view, which aims to provide rigorous foundations for areas such as programming. Dijkstra tried to reconcile both views. As a result, he contributed to both sides of the divide in a number of fundamental ways.

Dijkstra was also a most striking and unusual person. He was admired and criticized, in equal measure, and commented upon by almost everyone he came into contact with. Yet, despite his achievements, Dijkstra has always remained largely unknown outside computer science. Eighteen years after his death, few people have heard of him, even in his own country.

22.1 Biography

Edsger Dijkstra was born in Rotterdam in 1930. He described his father, at one time the president of the Dutch Chemical Society, as “an excellent chemist,” and his mother as “a brilliant mathematician who had no job.” EWD1166 [Dijkstra 1993]. In 1948, Dijkstra achieved remarkable results when he completed secondary school at the famous *Erasmiaans Gymnasium* in Rotterdam. His school diploma shows that he earned the highest possible grade in no less than six out of thirteen subjects. He then enrolled at the University of Leiden to study physics.

In September 1951, Dijkstra’s father suggested he attend a three-week course on programming in Cambridge. It turned out to be an idea with far-reaching consequences. It was in Cambridge that Dijkstra met the mathematician and computer scientist Adriaan van Wijngaarden, who subsequently offered him a job at the Mathematisch Centrum (Mathematical Centre) in Amsterdam, which he joined the following year. Dijkstra became, in his own words, “the first Dutchman with the qualification ‘programmer’ on the payroll.” EWD1166 [Dijkstra 1993]. In 1956, he finished his studies in Leiden. Three years later, he defended his PhD thesis, “Communication with an Automatic Computer.” His supervisor was van Wijngaarden.

Dijkstra worked at the Mathematisch Centrum until 1962, when he moved to Eindhoven to assume the position of professor in the mathematics department of the Eindhoven University of Technology. In 1973, he reduced his employment to one day a week and for the remaining four days worked as a research fellow at the Burroughs Corporation, at that time an innovative American computer manufacturer. His only duties for Burroughs involved undertaking research and traveling to the US a few times each year to visit the company headquarters.

In Dijkstra's reports, he listed the address Plataanstraat 5, Nuenen 4565, The Netherlands. This led some to assume that the Burroughs Corporation had opened a new office. The address was, in fact, that of Dijkstra's home, a modest house situated in a village near the outskirts of Eindhoven. His office consisted of a small room on the second floor, which was equipped with an "elegant portable Olivetti typewriter" and "two telephones . . . one of which he could use to call anywhere in the world, with the bills going direct to Burroughs." [van Emden 2008].

In 1984, disenchanted with a change of direction at the Burroughs Corporation and a lack of support for computer science at his university, Dijkstra left the Netherlands and took up a prestigious chair in computer science at the University of Texas at Austin. "Whereas Burroughs's intellectual horizon was shrinking," he later wrote, "my own was widening." EWD1166 [Dijkstra 1993]. He retired in 1999.

In early 2002, Dijkstra learned that he was terminally ill and moved back to Nuenen with his wife, Ria. He passed away in August, just a few months after returning. Ria died ten years later. The couple are survived by three children: Femke, Marcus, and Rutger.

Over the course of his career, Dijkstra wrote around 40 journal publications and 30 conference publications.[1] He is listed as the sole author for almost all of these works. Several of his journal papers were just a couple of pages long, while most of his conference publications were nonrefereed manuscripts that he presented during the Biannual International Marktoberdorf Summer School and published in the school proceedings. He also wrote a handful of book chapters and a few books.

Viewed from this perspective, Dijkstra's research output appears respectable, but otherwise unremarkable by current standards. In this case, appearances are indeed deceptive. Judging his body of work in this manner misses the mark completely. Dijkstra was, in fact, a highly prolific writer, albeit in an unusual way.

22.2 EWDs

In 1959, Dijkstra began writing a series of private reports. Consecutively numbered and with his initials as a prefix, they became known as EWDs. He continued writing these reports for more than forty years. The final EWD, number 1,318, is dated April 14, 2002. In total, the EWDs amount to over 7,700 pages. Each report was photocopied by Dijkstra himself and mailed to other computer scientists. The recipients varied depending on the subject. Around 20 copies of each EWD were distributed in this manner.

[1] An almost complete list of his publications can be found in the on-line Computer Science bibliography at https://dblp.uni-trier.de/.

The EWDs were initially written in Dutch using a typewriter. In 1972, Dijkstra switched to writing exclusively in English, and in 1979 he began writing them mostly by hand. The EWDs consisted of research papers, proofs of new or existing theorems, comments or opinions on the scientific work of others (usually critical and occasionally titled "A somewhat open letter to . . . "), position papers, transcripts of speeches, suggestions on how to conduct research ("Do only what only you can do"), opinions on the role of education and universities ("It is not the task of the University to offer what society asks for, but to give what society needs" EWD1305 [Dijkstra 2000], and original solutions to puzzles. Later reports also included occasional accounts of Dijkstra's life and work. A number of EWDs are titled "Trip Report" and provide detailed descriptions of his travels to conferences ("I managed to visit Moscow without being dragged to the Kremlin" EWD1166 [Dijkstra 1993]), summer schools, or vacation destinations. These reports are a rich source of information about Dijkstra's habits, views, thinking, and (hand)writing. Only a small portion of the EWDs concerned with research ever appeared in scientific journals or books.

This way of reporting research was, in fact, common during the eighteenth century. In the twentieth century it was a disarming anachronism. Nevertheless, it worked. In EWD1000 [Dijkstra 1987], dated January 11, 1987, Dijkstra recounts being told by readers that they possessed a sixth or seventh generation copy of EWD249.

Whether written using a fountain pen or typewriter, Dijkstra's technical reports were composed at a speed of around three words per minute. "The rest of the time," he remarked, "is taken up by thinking." EWD1000 [Dijkstra 1987]. For Dijkstra, writing and thinking blended into one activity. When preparing a new EWD, he always sought to produce the final version from the outset.

Around 1999, Hamilton Richards, a former colleague of Dijkstra's in Austin, created a website to preserve all the available EWDs and their bibliographic entries.[2] The E. W. Dijkstra Archive, as the site is known, also offers an abundance of additional material about Dijkstra, including links to scans of his early technical reports, interviews, videos, obituaries, articles, and a blog.

Despite a worldwide search, a number of EWDs from the period prior to 1968 have never been found. Other missing entries in the numbering scheme were, by Dijkstra's own admission, "occupied by documents that I failed to complete." [Dijkstra 1982b].

[2] E. W. Dijkstra Archive: The Manuscripts of Edsger W. Dijkstra, 1930–2002. The original EWD manuscripts are housed at the Dolph Briscoe Center for American History at the University of Texas at Austin.

22.3 Early Contributions

Dijkstra was a true pioneer in his field. This occasionally caused him problems in everyday life. In his Turing Award lecture he recalled:

> [I]n 1957, I married, and Dutch marriage rites require you to state your profession and I stated that I was a programmer. But the municipal authorities of the town of Amsterdam did not accept it on the grounds that there was no such profession [Dijkstra 1972a].

In the mid-1950s, Dijkstra conceived an elegant shortest path algorithm. There were very few computer science journals at the time and finding somewhere to publish his three-page report proved far from easy. Eventually, three years later, he settled on the newly established *Numerische Mathematik* [Dijkstra 1959]. "A Note on Two Problems in Connexion with Graphs" remains one of the most highly cited papers in computer science, while Dijkstra's algorithm has become indispensable in GPS navigation systems for computing the shortest route between two locations.

Over a period of eight months beginning in December 1959, Dijkstra wrote an ALGOL 60 compiler with Jaap Zonneveld.[3] Theirs was the first compiler for this new and highly innovative programming language. It was a remarkable achievement. In order to write the compiler, several challenges had to be overcome. The most obvious problem the pair faced was that the machine designated to run the software, the Dutch Electrologica X1 computer, had a memory of only 4,096 words. By comparison, the memory of a present-day laptop is larger by a factor of a million.

The programming language itself was not without its own challenges. ALGOL 60 included a series of novel features, such as recursion, which was supported in a much more complex manner than logicians had ever envisaged.[4] One of the ideas suggested by Dijkstra, termed a display, addressed the implementation of recursive procedures and has since become a standard technique in compiler writing [Dijkstra 1960].

ALGOL 60 was designed by an international committee. Although Dijkstra attended several meetings during the design process, his name does not appear among the thirteen editors of the final report [Backus et al. 1960]. Apparently, he

[3] A compiler is a computer program that translates a given source program into a low-level language, so that it can be executed directly by the used computer.

[4] Recursion refers to the property that a function or a procedure is defined in terms of itself. It was first introduced in programming languages a couple of months earlier by John McCarthy in the language Lisp. Recursive procedures in ALGOL 60 are much more complex than in Lisp.

disagreed with a number of majority opinions and withdrew from the committee. This was perhaps the first public sign of his fiercely held independence.

During his employment at the Eindhoven University of Technology, Dijkstra and his group wrote an operating system for the Electrologica X8, the successor to the X1. The system they created, known as the THE multiprogramming system (THE is an abbreviation of Technische Hogeschool Eindhoven), had an innovative layered functional structure, in which the higher layers depended only on the lower ones [Dijkstra 1968a].

It was during his work on this system that Dijkstra's interests began shifting to parallel programs, of which THE is an early example. These programs consist of a collection of components, each of which is a traditional program, executed in parallel. Such programs are notoriously difficult to write and analyze because they need to work correctly no matter the execution speeds of their components. Parallel programs also need to synchronize their actions to ensure exclusive access to resources. If several print jobs are dispatched at the same time by the users of a shared computer network, this should not lead to pages from the different print jobs becoming interspersed. Adding to the complexity, the components of parallel programs should not become deadlocked, waiting indefinitely for one another.

In the early 1960s, these problems had not yet been properly examined or analyzed, nor had any techniques been developed to verify potential solutions. Dijkstra identified a crucial synchronization problem, which he named the mutual exclusion problem, and published his solution in a single-page paper [Dijkstra 1965]. This work was taken from EWD123, an extensive 87-page report titled "Cooperating Sequential Processes." In the same report, he introduced the first known synchronization primitive, which he termed a semaphore, that led to a much simpler solution to the mutual exclusion problem.[5] He also identified the deadlock problem, which he named the deadly embrace, and proposed an elegant solution, the banker's algorithm [Dijkstra 1968c].[6] The mutual exclusion problem, along with deadlock detection and prevention, are now mandatory topics in courses on operating systems and parallel programming.

In 1968, Dijkstra published a two-page letter addressed to the editor of the *Communications of the ACM*, in which he critiqued the goto programming statement [Dijkstra 1968b].[7] Entitled "Go To Statement Considered Harmful," the letter was

[5] The mutual exclusion problem and semaphores were originally proposed in 1962 in EWD35 "Over de sequentialiteit van procesbeschrijvingen," written in Dutch.

[6] The banker's algorithm was originally proposed in 1964 in EWD108 "Een algorithme ter voorkoming van de dodelijke omarming", written in Dutch.

[7] Dijkstra had, in fact, used the goto statement in an article three years earlier, [Dijkstra 1965].

widely criticized and generated considerable debate. In the end, Dijkstra's views prevailed. Every programmer is now aware that using the goto statement leads to so-called spaghetti code. Java, currently one of the most widely used programming languages, was originally released in 1996 and does not have the goto statement. The phrase "considered harmful" is still used often in computer science and remains inextricably associated with Dijkstra.

In 1968, Dijkstra suffered a long, deep depression that persisted for almost half a year. He later made mention of being hospitalized during this period [Dijkstra 1987]. One reason for Dijkstra's torment was that his department did not consider computer science important and disbanded his group. He also had to decide what to work on next. Dijkstra's major software projects, the ALGOL 60 compiler and the THE multiprogramming system, had given him a sense that programming was an activity with its own rules. He then attempted to discover those rules and present them in a meaningful way. Above all, he strove to transform programming into a mathematical discipline, an endeavor that kept him busy for several years to come. At the time, these were completely uncharted waters. Nobody else seemed to be devoting their attention to such matters.

A year later, the appearance of the 87-page EWD249, "Notes on Structured Programming," marked the end of Dijkstra's depression [Dijkstra 1972b]. The subject of the EWD was so novel, the writing so engaging, and the new term "structured programming" so convincing that the report became a huge success. But, in Dijkstra's view, "IBM . . . stole the term 'Structured Programming' and . . . trivialized the original concept to the abolishment of the goto statement." [Dijkstra 2002b]. The claim was unsurprising to those aware of Dijkstra's long-held and largely negative views toward IBM computers and software. Nowadays it is not uncommon to see similar criticisms of large corporations, but in the 1970s and 1980s few academics were prepared to take a public stand against computer manufacturers.

In 1972, Dijkstra received the ACM Turing Award, widely considered the most important prize in computer science. He was recognized for

> fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design.[8]

Further fundamental contributions were to follow. In 1974, Dijkstra published a two-page article in which he introduced a new concept: self-stabilization

[8] Citation: Edsger Wybe Dijkstra," ACM A. M. Turing Award

[Dijkstra 1974a]. In the paper, he posed the problem of how a system of communicating machines might repair itself when a temporary fault arises in one of the machines. He presented new protocols that guaranteed correct functioning of the system would eventually be restored. "Self-stabilization," he remarked, " . . . could be of relevance on a scale ranging from a *worldwide network* [emphasis added] to common bus control." This is a striking observation when one considers that the World Wide Web was developed just 15 years later. As it turned out, the paper was completely ignored until 1983, when Leslie Lamport stressed its importance in an invited talk. In time, the ideas outlined by Dijkstra would lead to the emergence of a whole new area in distributed computing with its own annual workshops and conferences. In 2002, the paper won an award that was posthumously renamed the Edsger W. Dijkstra Prize in Distributed Computing.

The notion that some events cannot be deterministically predicted, usually referred to as indeterminism, has kept philosophers, and later physicists, occupied for centuries. Computer scientists enter the story more recently, studying the idea under the name nondeterminism — not a reference, it should be noted, to any probabilistic interpretation of events. In 1963, John McCarthy introduced nondeterminism in the context of programming languages. A couple of years later, Robert Floyd showed how this concept, now known as angelic nondeterminism, can substantially simplify programming tasks requiring a search [Floyd 1967, McCarthy 1963]. When choices arise there is *some* computation that delivers the desired result —though it is not certain which one.

Dijkstra's view of nondeterminism was likely influenced by the inherently nondeterministic behavior of the real-time interrupt handler he developed in his PhD thesis. In a 1975 paper, he introduced a small programming language that he called guarded commands; it encapsulated what is now termed demonic nondeterminism [Dijkstra 1975]. This was, in fact, the paper he handed me the first time I met him. In contrast to the angelic variant, *all* computations have to deliver the desired result. This more demanding view of nondeterminism —referred to as *nondeterminacy* by Dijkstra —sometimes yields simpler programs, but for reasons other than angelic nondeterminism. The programmer is free to leave some decisions unspecified.

The programming notation introduced by Dijkstra occasionally leads to elegant programs. He illustrated this point by reconsidering Euclid's 2,300-year-old algorithm for computing the greatest common divisor of two natural numbers. The algorithm can be stated as follows: as long as the two numbers differ, keep subtracting the smaller number from the bigger one. In Dijkstra's language this algorithm can be written in a simple manner that is not far removed from its description in English.[9]

His language also introduced the crucial notion of a *guard*, which has since become a natural concept in various programming formalisms. Similarly, *weakest precondition semantics*, a concept Dijkstra introduced to describe program semantics, marked a late but highly significant entry into the field of program verification. A couple of years later, the language was generalized by Tony Hoare to create a highly influential programming language proposal for distributed computing that he named CSP [Hoare 1978].

Dijkstra's landmark book *A Discipline of Programming* was published in 1976 [Dijkstra 1976]. It introduced a novel approach to programming in which Dijkstra combined weakest precondition semantics with a number of heuristics to develop several computer programs, hand in hand with their correctness proofs. In contrast with EWD249, "Notes on Structured Programming," he was now arguing about program correctness in a formal way. This development marked a new stage in Dijkstra's research. He now viewed the development of a correct program as the development of a mathematical proof, something to which he first alluded in 1973 as part of EWD361, "Programming as a Discipline of Mathematical Nature." [Dijkstra 1974b]. This methodology was soon employed by Dijkstra and a group of researchers to systematically derive various, usually small, nontrivial programs. In contrast to some of his other innovations, it never really caught on.

22.4 Eighties

In the early 1980s, Dijkstra cowrote two short but influential papers in which he applied his methodology to the systematic development of distributed programs [Dijkstra and Scholten 1980, Dijkstra et al. 1983]. He also sought to have this approach to programming taught to first-year students, and, with this goal in mind, put together an elegant introductory textbook with Wim Feijen [Dijkstra and Feijen 1988].

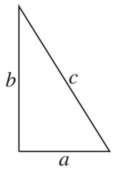
[9] Dijkstra's solution is the following simple program:

$$\mathbf{do}\ x > y \rightarrow x := x - y\ [\!]\ y > x \rightarrow y := y - x\ \mathbf{od}.$$

Here the initial values are stored in the variables x and y, $[\!]$ represents the nondeterministic choice, := should be read as "becomes", and the **do** ... **od** construct represents a repetition. Further, $x > y$ and $y > x$ are so-called guards, conditions under which the actions that follow can be carried out.

Dijkstra's realization that programming could be viewed as a mathematical activity led to his interest in analyzing mathematical reasoning. He attempted to come up with guidelines and heuristics that facilitated the discovery of proofs. In a number of cases these principles pointed him toward interesting generalizations of known results that had somehow eluded others.

A good example is the Pythagorean theorem, which is taught in secondary schools. The theorem states that in a right-angled triangle the square of the hypotenuse, c, is equal to the sum of the squares of the other two sides, a and b.

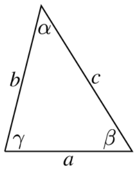


$a^2 + b^2 = c^2$.

In 1940, Elisha Loomis collected no less than 370 proofs in *The Pythagorean Proposition*, starting with the proof that appeared in Euclid's *Elements*, written around 300 BCE [Loomis 1940]. New proofs occasionally appear to this day.[10]

In 1986, Dijkstra came up with the following generalization to arbitrary triangles, which he included in EWD975:

> Consider a triangle with the side lengths a, b and c and the angles α, β and γ, lying opposite a, b and c. Then the signs of the expressions $a^2 + b^2 - c^2$ and $\alpha + \beta - \gamma$ are the same (that is, they are either both positive or both zero or both negative).



$sign(a^2 + b^2 - c^2) = sign(\alpha + \beta - \gamma)$.

A mathematician might observe that this all seems quite obvious. Yet, apparently, nobody had thought of this generalization before Dijkstra. He concluded his report by observing that it was unclear where he might publish this result. In his view, it should be taught at schools instead of the original theorem. In 2009, EWD975 was republished posthumously by *Nieuw Archief voor Wiskunde*

[10] An example of a new proof is [Basu 2015].

(New Archive for Mathematics), the magazine of the Royal Dutch Mathematical Association [Dijkstra 2009]. The five-page report was reproduced in its original handwritten form.

In a 1985 lecture, "On Anthropomorphism in Science," delivered at the University of Texas at Austin's Philosophy Department, Dijkstra speculated that mathematicians stuck to the use of implication because they associated it with cause and effect. "Somehow," he observed, "in the implication 'if A then B,' the antecedent A is associated with the cause and the consequent B with the effect." EWD936. He claimed that equivalence should be preferred over implication. This simple principle had led to his generalization of the Pythagorean theorem.

Dijkstra also attempted to apply his methodology for developing correct programs to systematically develop proofs of mathematical theorems. In EWD1016, "A Computing Scientist's Approach to a Once-Deep Theorem of Sylvester's," he derived an elegant proof of the following theorem, first conjectured in 1893 and proved 40 years later: "Consider a finite number of distinct points in the real Euclidean plane; these points are collinear or there exists a straight line through exactly 2 of them." [Dijkstra 1988b].

Another example from this period is his approach to the problem of a fair coin. A coin toss is used to determine one of two outcomes in a fair way. But how can a fair outcome be achieved when the coin is biased? In 1951, John von Neumann came up with a simple solution [von Neumann 1951]. A number of researchers then tried to figure out how to make it more efficient. Dijkstra first learned of the problem during a lecture in 1989. He solved it immediately and a little while later came up with the solution to a related problem that apparently nobody had thought of before. His solution to the related problem can be found in EWD1071, "Making a Fair Roulette from a Possibly Biased Coin." Dijkstra's modification of von Neumann's solution was not immediately obvious and relies on a classic result from number theory, Fermat's little theorem. For a change, Dijkstra submitted this article to a journal and it was published the following year as a one-page note [Dijkstra 1990].

The development of a natural and readable notation for representing proofs and calculations was almost as important for Dijkstra as the problems under consideration. The notation he came up with forces the author not to commit what he described as "any sins of omission":

$$\begin{array}{lll} & A & \\ \longrightarrow & & \{\text{hint why } A \longrightarrow B\} \\ & B & \\ \longrightarrow & & \{\text{hint why } B \longrightarrow C\} \\ & C & \end{array}$$

In his final EWD, EWD1318 [Dijkstra 2002c] for example, Dijkstra explains that for $s = (a+b+c)/2$ the equality $s(s-b)(s-c)+s(s-c)(s-a) = s(s-c)c$ holds, by writing out his argument as:

$$\begin{array}{ll} & s(s-b)(s-c) + s(s-c)(s-a) \\ = & \quad \{\text{algebra}\} \\ & s(s-c)(2s-a-b) \\ = & \quad \{\text{definition of } s\} \\ & s(s-c)c \end{array}$$

He used this notation in his own publications, notably in a book he wrote with his longstanding friend and colleague Carel Scholten [Dijkstra and Scholten 1990]. The notation was adopted by several of his colleagues but did not spread further. EWD1300, "The Notational Conventions I Adopted, and Why," was republished posthumously and offers a unique insight into Dijkstra's extensive work on notation, a subject that kept him busy throughout his career [Dijkstra 2002a].

In the late 1980s, Dijkstra's research was described on his departmental homepage as follows: "My area of interest focuses on the streamlining of the mathematical argument so as to increase our powers of reasoning, in particular, by the use of formal techniques." It was also the subject of his course for computer science students.

During the period 1987 – 1990, I was a fellow faculty member in Austin and followed his course for a semester. Dijkstra invariably arrived early for class so that he could write out an unusual quotation on the blackboard. The lectures themselves were always meticulously prepared and usually devoted to presenting proofs of simple mathematical results. He delivered the lectures without notes, requiring only a blackboard and a piece of chalk. At the end of each lecture he would assign an elementary but nontrivial mathematical problem as homework and collect all the solutions at the following lecture. A week later he would return all the solutions with detailed comments, sometimes longer than the actual submissions, and then present his own solution in detail, stressing the presentation and use of notation. My own solutions fared reasonably well by Dijkstra's standards and were usually returned with only short comments, such as "Many sins of omission." It was, in fact, a course in orderly mathematical thinking, and nobody seemed at all bothered that it had nothing to do with computer science.

Dijkstra was a highly engaging lecturer. He knew how to captivate an audience with dramatic pauses, well-conceived remarks, and striking turns of phrase. The bigger the audience, the better he performed. While I was working in Austin I helped organize a departmental event with him as the main speaker. About two hundred people came along, some having flown in from Houston and neighboring states to attend the event. Dijkstra stole the show and delivered a mesmerizing presentation on Sylvester's theorem.

22.5 Nineties

In 1990, Dijkstra's sixtieth birthday was celebrated in Austin with a large banquet featuring a distinguished group of guests, including numerous important figures in computer science. A Festschrift was published by Springer-Verlag to mark the occasion. The volume began on page 0, in deference to the way Dijkstra numbered the pages of his EWDs. He took the trouble of thanking each of the 61 contributing authors by a handwritten letter.

The period that followed was marked by a visible change in Dijkstra's attitude and approach to his work. In the remaining twelve years of his life, despite producing about 250 new EWDs, he published almost nothing. These reports simply may not have met his standards for a journal publication. Many of the EWDs were devoted to systematically deriving proofs of tricky results, such as a problem from the International Mathematical Olympiad. He also used his methodology to obtain elegant solutions for classical puzzles, such as the knight's tour or the wolf, goat, and cabbage puzzle.[11] Some of the EWDs from this period contained accounts and assessments of his early contributions.

Following Dijkstra's retirement from teaching in the fall of 1999, a symposium was organized in May 2000 to honor his seventieth birthday. The event was called "In Pursuit of Simplicity" and included guest contributors from both Europe and the US. At this time, I was working at the Centre for Mathematics and Informatics (CWI) in Amsterdam and during the symposium I invited Dijkstra to give a lecture. CWI was, in fact, the new name for the Mathematisch Centrum where he had worked at the beginning of his career. Six months later Dijkstra accepted the invitation. He had never seen the new and larger building where the CWI had relocated in the early 1980s, and was visibly moved.

Prior to the lecture, the CWI's communication department issued a press release. News of the event caught the attention of a major Dutch newspaper. A journalist was dispatched and an extensive article with a prominent photo of Dijkstra was soon published. VPRO, an independent Dutch public broadcasting company, subsequently became interested in Dijkstra and sent a crew to Austin to make a

[11] Regrettably, nobody convinced Dijkstra to select some of these masterful expositions and publish them as a book [Dijkstra 1992, 1995, 1997]. The knight's tour problem involves finding a knight's path on the chess- board that visits each square exactly once.

half-hour-long program about him. “Denken als discipline” (Thinking as a Discipline) was broadcast in April 2001 as an episode of the science show Noorderlicht (Northern Lights).[12] The episode received a glowing review in another prominent Dutch newspaper.

In early 2002, Dijkstra returned to Nuenen, incurably ill with cancer. The news spread quickly in the computer science community and was invariably met with deep sadness. The last time I saw Dijkstra was at his home in July 2002. As was usually the case with visitors, he collected me by car from the nearest train station a couple of kilometers away from his house. During the visit, we spoke together, shared lunch, and he told me that he did not have much time left. He also gave me a copy of EWD1318, “Coxeter’s Rabbit,” dated April 14, 2002, mentioning that it would be his final report.[13]

Dijkstra passed away a month later. His funeral was attended by a number of his colleagues, including several from Austin. In his eulogy, Hoare reflected on Dijkstra’s immense contributions to the development of his field:

> He would lay the foundations that would establish computing as a rigorous scientific discipline; and in his research and in his teaching and in his writing, he would pursue perfection to the exclusion of all other concerns. From these commitments he never deviated, and that is how he has made to his chosen subject of study the greatest contribution that any one person could make in any one lifetime [Memoriam].

J Strother Moore, then chairman of the computer science department in Austin, also spoke warmly and evocatively of Dijkstra: “He was like a man with a light in the darkness. He illuminated virtually every issue he discussed.” [Memoriam].

Obituaries subsequently were published in several leading newspapers, including the New York Times, the Washington Post, and The Guardian. Extended commemorative pieces and reminiscences appeared during the months that followed, in which Dijkstra was variously acclaimed as a pioneer, prophet, sage, and genius [Apt 2002, Boyer et al. 2002, van Emden 2008].

22.6 Opinions and Attitudes

Dijkstra’s enduring influence in computer science is not confined solely to his research. He held strong opinions about many aspects of the field, most notably

[12] The EWD website has a copy of this video with subtitles in English. It is still worthwhile to watch it today, nineteen years later. Only then can one better appreciate Dijkstra’s unique mixture of precision, modesty, charm, and self-assuredness, with an utmost focus on whatever kept him busy at that moment.

[13] To my surprise Richards did not have it and quickly updated the EWD website.

about programming languages and the teaching of programming, but also the purposes of education and research.

Dijkstra did not shy away from controversies. He was a dedicated contrarian who reveled in expressing extreme and unconventional opinions. I saw this first-hand in 1977 during a large computer science conference in Toronto. Audiences for plenary lectures at the event numbered somewhere between several hundred and a thousand attendees. Each of the lectures concluded with a few polite audience questions for the speaker, generally a leading expert in his area. I have a vivid recollection of Dijkstra standing up at the end of one lecture and delivering a stinging rebuke to the speaker. Contrary to appearances, he was hoping to provoke an informative discussion. The chairman was visibly taken aback by Dijkstra's intervention and appeared at a loss as to how he should proceed.

Dijkstra was also unafraid to voice harsh critiques at smaller gatherings. In the late 1970s, I attended a seminar in Utrecht with about twenty other participants, including Dijkstra. He repeatedly interrupted a highly respected lecturer to query him about his use of terminology and abbreviations. Halfway through the presentation Dijkstra abruptly got up and left. Other stories in a similar vein were far from uncommon and circulated throughout the field.

Dijkstra took his work as a reviewer extremely seriously and his reports were detailed and carefully thought out. Some of these reviews were undertaken at his own initiative and appeared as EWDs. These included a positive review of a 400-page computer science book that had no obvious connection to his research [Dijkstra n.d.]. He also produced extensive and thoughtful comments on manuscripts sent to him by his colleagues who had adopted his notation or methodology, or whose research he deemed important.

In 1977, Dijkstra wrote a vitriolic review of a report, "Social Processes and Proofs of Theorems and Programs," by Richard De Millo, Richard Lipton, and Alan Perlis. The report later appeared as a journal paper [DeMillo et al. 1979]. Dijkstra distributed his review as EWD638, "A Political Pamphlet from the Middle Ages," in which he referred to the report as "a very ugly paper." [Dijkstra 1978b]. The authors had argued that "program verification is bound to fail," a view Dijkstra vehemently disagreed with.

Some of these reviews led to further correspondence with the original authors. In 1978, Dijkstra distributed a detailed and scathing review of the 1977 Turing Award Lecture delivered by John Backus. In EWD692 he argued that the lecture "suffers badly from aggressive overselling." [Dijkstra 1978a]. At the time, Backus was one of the most prominent working computer scientists. He was the co-inventor of a standard notation used to describe the syntax of programming languages, known as Backus-Naur form, and had led the team that designed and implemented Fortran, the first high-level programming language. In his lecture, Backus had advocated for an alternative style of programming, known as functional programming. Four years ago, Jiahao Chen discovered an extensive and highly critical correspondence between Backus and Dijkstra that took place following the review.[14] Dijkstra wisely kept these exchanges away from the public eye.

In some quarters, Dijkstra was viewed as arrogant and his opinions considered extreme. When cataloguing their correspondence in his papers, Backus added the comment: "This guy's arrogance takes your breath away." [Chen 2016]. For many, especially those who adopted his notation, Dijkstra became a figure akin to a guru. A small group of his followers even started their own EWD-like reports, all consecutively numbered and written by hand. Dijkstra seemed indifferent to such displays. "But he takes himself so seriously," a Turing Award winner once confided to me. Indeed, one sometimes had the impression that he carried the weight of computer science on his shoulders.

In 1984, I was invited to be a lecturer at the annual Marktoberdorf School, co-organized by Dijkstra. Although I regarded this as a great honor, I could not help but feel anxious about having him assessing my work. His review appeared in EWD895, "Trip report E. W. Dijkstra, Marktoberdorf, 30 July - 12 Aug. 1984." I was greatly relieved to discover his comments were not only fairly mild, but even reasonably positive in comparison to his other reviews: "Apt's lectures suffered somewhat from this [i.e., talking 'exclusively about CSP'] . . . The examples chosen to illustrate his points were a bit elaborate, but his conscious efforts to be understood were highly appreciated." [Dijkstra 1984].

Encouraged by this appraisal, I submitted one of the papers I had presented to a peer-reviewed journal. A couple of months later, the anonymous referee reports arrived. One of the reviews was unmistakably the work of Dijkstra. A detailed criticism of what he regarded as a lack of sufficiently formal arguments, combined with a long list of demands and questions, made attempting satisfactory revisions a hopeless task. Even today, I would not know how to meet these demands because the right formalism is still lacking. At the time, it was nonetheless considered a privilege to have a paper refereed by Dijkstra.

In 1989, Dijkstra presented his views on teaching computer programming in a lecture titled "On the Cruelty of Really Teaching Computer Science" during an ACM Computer Science Conference. A transcript was later circulated as EWD1036 [Dijkstra 1988a]. After presenting a sweeping historical survey aimed at illustrating traditional resistance toward new ideas in science, Dijkstra argued that computer

[14] In particular, Backus considered it rude that Dijkstra sent this EWD to others, but not to him [Chen 2016].

programming should be taught in a radically different way. His proposal was to teach some of the elements of mathematical logic, select a small but powerful programming language, and then concentrate on the task of constructing provably correct computer programs. In his view, programs should be considered the same way as formulas, while programming should be taught as a branch of mathematics. There was no place for running programs or for testing, both of which were considered standard practice in software engineering.

The lecture and report led to an extensive debate that still makes for interesting reading. Dijkstra's report was published in Communications of the ACM, along with his polite but unapologetic responses to mostly negative reactions from prominent computer scientists. While he was praised for initiating a much-needed debate, Dijkstra's recommendations were deemed unrealistic and too controversial [Denning 1989].

Dijkstra often expressed his opinions using memorable turns of phrase or maxims that caught the ears of his colleagues and were widely commented upon. Here are some examples:

- Program testing can be used to show the presence of bugs, but never to show their absence.
- Computer science is no more about computers than astronomy is about telescopes.
- The question of whether machines can think is about as relevant as the question of whether submarines can swim.
- A formula is worth a thousand pictures.

In one of his EWDs, Dijkstra collected several jibes about programming languages, such as: "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense." [Dijkstra 1982a]. At the time, COBOL was one of the most widely used programming languages and these comments were not warmly received.

Some of Dijkstra's opinions were unavoidably controversial and highlighted his longstanding prejudices. When I first met him in 1975 he recommended the book *Structured Programming*, but suggested that I skip the final chapter by Hoare and Ole-Johan Dahl, [Dahl and Hoare 1972], as it dealt with object-oriented programming. Nonetheless, object-oriented programming gradually became a universally preferred way to structure a large program. But not for Dijkstra. He was still arguing against it in 1999, pointing out during a keynote address that, "For those who have wondered: I don't think object-oriented programming is a structuring paradigm that meets my standards of elegance." [Dijkstra 1999]. By that time, the popular object-oriented programming language C++ was routinely taught to first-year computer science students.

22.7 Character and Lifestyle

Throughout his professional career, Dijkstra remained remarkably modest. He never had a secretary; he typed or wrote all his publications himself. Most were entirely his own work and even the few that listed coauthors were clearly written by Dijkstra, or in his style. After 1979, he preferred to write by hand using a Montblanc fountain pen. His writing style became so recognizable among computer scientists in the 1980s that a fellow academic, Luca Cardelli, designed a Dijkstra font for Macintosh computers.[15] Not long after it was released, Dijkstra received a letter typeset in Cardelli's font and mistakenly assumed it was handwritten. He felt tricked by the letter and was not amused. Some years later, he was able to appreciate the humor when a colleague in Austin, Bob Boyer, adopted the font for presentations during departmental meetings.

It seems Dijkstra never applied for any grants —though he did receive at least one, to employ a PhD student— nor did he bring any money, in the form of research contracts, to the institutions he worked for. He also never purchased a computer. Eventually, in the late 1980s, he was given one as a gift by a computer company, but never used it for word processing. Dijkstra did not own a TV, a VCR, or even a mobile phone. He preferred to avoid the cinema, citing an oversensitivity to visual input. By contrast, he enjoyed attending classical music concerts.

When taking part in conferences and summer schools Dijkstra often felt uncomfortable in large groups. Unaccustomed to small talk, he usually remained awkwardly silent. Away from the work environment, however, he was completely different. From his time in Austin, I and others recall him as friendly, helpful, and eager to drop by with his wife for a short social visit that often led to engaging conversations. He and his wife liked to invite guests over, for whom he occasionally played short pieces of classical music on his Bösendorfer piano. His favorite composer was Mozart. A striking feature of Dijkstra's living room was a lectern with a large copy of the *Oxford English Dictionary*, which he found indispensable in his work. He is, incidentally, mentioned in the same dictionary in connection with the use of the words *vector* and *stack* in computing.

In Austin, Dijkstra stayed away from university politics. He was highly respected by colleagues, not least because of his collegial attitude during departmental meetings. He took his teaching duties very seriously. Exams were always oral and could

[15] Luca Cardelli, Personal website.

last a couple of hours. Upon completion of the exam, an informal chat followed during which the student was presented with a signed photo of Dijkstra, and a beer —age permitting [Memoriam]. He held his weekly seminars in his office and served coffee to the students in attendance, often surprising them with his unassuming behavior.

Throughout his life, Dijkstra was an uncompromising perfectionist, always focused on tapping his creativity, unwilling to lower his standards, and indifferent toward alternative points of view. He also found it difficult to browse articles in his field to form an idea of their contents and seemed uninterested in tracking down and studying the relevant literature. For the most part, he followed the recommendations of his close colleagues and only studied the papers they suggested. As a result, his own papers often had very few, if any, bibliographic entries. The preface of *A Discipline of Programming* concludes with a frank admission: "For the absence of a bibliography I offer neither explanation nor apology." This cavalier approach led to occasional complaints from colleagues who found that their work was ignored.

Instead, Dijkstra preferred to study classic texts, such as Eric Temple Bell's *Men of Mathematics*, which he referred to occasionally during his courses in Austin, and Linus Pauling's *General Chemistry*, a book he praised in highest terms.[16] This attitude served him well during the 1960s and 1970s, but it became increasingly impractical as computer science matured.

Published in 1990, Dijkstra's *Predicate Calculus and Program Semantics*, cowritten with Scholten, was a case in point. The book not only lacked references, but also exhibited a complete disregard for the work of logicians. Egon Börger penned an extensive and devastating review, claiming that the approach outlined by the authors was not in any way novel, nor did it offer any advantages [Börger 1994]. He also vigorously criticized the book's rudimentary history of predicate logic, in which the authors drew a line from the work of Gottfried Leibniz to that of George Boole and then to their own contributions, neglecting to mention anyone else.

In response to Börger's review, some colleagues tried to help by publishing papers that provided a useful logical assessment and clarification of Dijkstra and Scholten's approach based on their so-called calculational proofs. Dijkstra remained unrepentant. "I never felt obliged to placate the logicians," he remarked some years later in EWD1227 [Dijkstra 1995/1996]. "If however, [logicians] only get infuriated because I don't play my game according to their rules," he added,

[16] Eric Temple Bell, *Men of Mathematics: The Lives and Achievements of the Great Mathematicians from Zeno to Poincaré* (New York: Simon and Schuster, 1937). Linus Pauling, *General Chemistry* (Ann Arbor, MI: Edward Brothers Inc., 1944).

referring specifically to Börger's review, "I cannot resist the temptation to ignore their fury and to shrug my shoulders in the most polite manner."

Dijkstra's sense of humor was, at turns, wry and terse. I once asked him how many PhD students he had. "Two," he replied, before adding, "Einstein had none."[17] On another occasion, he wrote to me that "[redacted] strengthened the Department by leaving it." In Austin, together with his wife, he purchased a Volkswagen bus, dubbed the Touring Machine, which they used to explore national parks.[18]

Dijkstra was also extremely honest. He was always insistent, for example, that the first solution to the mutual exclusion problem was found by his colleague Th.J. Dekker. In EWD1308 he admitted that F.E.J. Kruseman Aretz "still found and repaired a number of errors [in the ALGOL 60 compiler] after I had left the Mathematical Centre in 1962," and that the phrase "considered harmful" was, in fact, invented by an editor of the *Communications of the ACM*, Niklaus Wirth. Dijkstra's contribution was originally titled "A Case against the GO TO Statement." In the same EWD, he also admitted completely missing the significance of Floyd and Hoare's initial contributions to program verification. "I was really slow" he lamented.

22.8 Legacy

Dijkstra left behind a remarkable array of notions and concepts that have withstood the test of time: the display, the mutual exclusion problem, the semaphore, deadly embrace, the banker's algorithm, the sleeping barber and the dining philosophers problems, self-stabilization, weakest precondition, guard, and structured programming.

His shortest path algorithm is taught to all students of computer science and operations research and is always referred to as Dijkstra's algorithm. Several years ago I saw it illustrated, under this name, by means of an interactive gadget with lights and buttons at the Science Centre Singapore.

Dijkstra and Zonneveld's ALGOL 60 compiler is rightly recognized as a milestone in the history of computer science —albeit more so in Europe than elsewhere. An entire PhD thesis was recently devoted to its reconstruction and a detailed analysis [van den Hove d'Ertsenryck 2019]. The layered design of the THE multiprogramming system is discussed in several textbooks on operating systems, and influenced the design of some later operating systems.

[17] Eventually, he had seven, if one counts joint supervisions.

[18] A Turing machine, invented by Alan Turing in 1936, is a mathematical model of a machine that can perform computations.

Among the concepts invented by Dijkstra, some have been reflected in book titles. An early example is *Structured Programming*, published in 1972 [Dahl et al. 1972]. There are now several books called *Structured Programming Using Language X*. In 1986, a book appeared with the title *Algorithms for Mutual Exclusion* [Raynal 1986] —others with a similar title eventually followed— and in 2000 a book titled *Self-Stabilization* was published [Dolev 2000].

Dijkstra's approaches to nondeterminism and parallelism are part of standard courses on these subjects. His proposal for a first synchronization mechanism triggered research that, in turn, resulted in the development of high-level synchronization mechanisms that are indispensable for parallel programming. His classic problems, such as the sleeping barber and dining philosophers, the latter named by Hoare, have become standard benchmarks. When methods were developed to formally verify parallel programs, the first examples tackled were Dijkstra's programs.

Yet Dijkstra's most enduring contribution may well be indirect —in software engineering. The challenge of producing correct software was an ongoing concern throughout his scientific career. In 1962, he wrote a paper, "Some Meditations on Advanced Programming," in which he raised the issue of program correctness and expressed the hope that this aspect might someday be referred to as the science of programming [Dijkstra 1962]. At a major conference in 1968, it was recognized that the availability of increasingly powerful computers led to increasingly complex and unreliable software systems, a problem termed "the software crisis." Dijkstra, who was in attendance, could not have agreed more.

In the years that followed, he produced a number of engaging and influential essays on software development in which he explicitly referred to the software crisis as an urgent problem. He forcefully argued that software systems should be built on sound design principles, and that correctness should be a driving principle behind program construction. In particular, he introduced the often-cited separation-of-concerns design principle, which, he remarked, "even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of." [Dijkstra 1982c]. Following the example of Hoare and Wirth, he also advocated for various forms of abstraction and the use of assertions to annotate programs. At a later stage, he argued that the programming process itself should be viewed as a mathematical activity.

Although Dijkstra's idealized vision that programs should be constructed together with their correctness proofs has not been realized, it undoubtedly provided the impetus for new methods of structuring and developing programs — including, somewhat paradoxically, the emergence of the object-oriented programming paradigm that he so vigorously opposed. This vision also helped motivate the design of new programming languages, along with platforms and systems to facilitate the programming process. Finally, Dijkstra's "Notes on Structured Programming" was highly influential in the development of better designed and more systematic courses on programming, occasionally with an emphasis on systematic program construction and correctness.

It is difficult to find another scientist who left such an impressive mark in the history of computer science.[19]

References

K. R. Apt. 2002. Edsger Wybe Dijkstra (1930-2002): A portrait of a genius. *Formal Aspects of Computing*, 14(2): 92–98. DOI: 10.1007/s001650200029.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and P. Naur (ed). 1960. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5): 299–314. https://doi.org/10.1145/367236.367262. DOI: 10.1145/367236.367262.

K. Basu, 2015. A new and very long proof of the Pythagoras theorem by way of a proposition on isosceles triangles. MPRA Paper 61125, University Library of Munich, Germany. Available at https://mpra.ub.uni-muenchen.de/61125/1/MPRA_paper_61125.pdf.

E. Börger. 1994. Book review E.W. Dijkstra, C.S. Scholten: Predicate Calculus and Program Semantics, Springer-Verlag 1989. *Science of Computer Programming*, 23(4): 1–11. DOI: 10.1016/0167-6423(94)90002-7.

R. S. Boyer, W. H. J. Feijen, D. Gries, C. A. R. Hoare, J. Misra, J. Moore, and H. Richards. 2002. In memoriam: Edsger W. Dijkstra 1930-2002. *Communications of the ACM*, 45(10): 21–22. DOI: 10.1145/570907.570921.

J. Chen, May 2016. This guy's arrogance takes your breath away. Letters between John W Backus and Edsger W Dijkstra, 1979. Available at https://medium.com/@acidflask/this-guys-arrogance-takes-your-breath-away-5b903624ca5f.

O.-J. Dahl and C. A. R. Hoare. 1972. Hierarchical program structures. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds., *Structured programming*, pp. 175–220. Academic Press Ltd.

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. 1972. *Structured programming*. Academic Press Ltd.

R. A. DeMillo, R. J. Lipton, and A. J. Perlis. 1979. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5): 271–280. https://doi.org/10.1145/359104.359106. DOI: 10.1145/359104.359106.

[19] Soon after Edsger's death, his wife, Ria Dijkstra, made me aware that many of Edsger's publications appeared much earlier as EWDs. The E. W. Dijkstra Archive was absolutely indispensable for writing this article. The memorial resolution [Memoriam] was very helpful. Jan Heering and Jay Misra helped me to understand in the past that Dijkstra's contributions to software construction and to nondeterminism were much more fundamental than I initially realized. Alma Apt, Maarten van Emden, and Jan Heering provided detailed comments on the initial version.

P. J. Denning. 1989. A debate on teaching computing science. *Communications of the ACM*, 32(12): 1397–1414. https://doi.org/10.1145/76380.76381. DOI: 10.1145/76380.76381.

E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271. DOI: 10.1007/bf01386390.

E. W. Dijkstra. 1960. Recursive programming. *Numerische Mathematik*, 2: 312–318. DOI: 10.1007/bf01386232.

E. W. Dijkstra. 1962. Some meditations on advanced programming. In *Information Processing, Proceedings of the 2nd IFIP Congress*, pp. 535–538. North-Holland. originally published as EWD32 in 1962.

E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9): 569. https://doi.org/10.1145/365559.365617. DOI: 10.1145/365559.365617.

E. W. Dijkstra. 1968a. The structure of "THE"-multiprogramming system. *Communications of the ACM*, 11(5): 341–346. DOI: 10.1145/363095.363143.

E. W. Dijkstra. 1968b. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3): 147–148. DOI: 10.1145/362929.362947.

E. W. Dijkstra. 1968c. Cooperating sequential processes. In F. Genuys, ed., *Programming Languages: NATO Advanced Study Institute*, pp. 43–112. Academic Press Ltd., London. originally published as EWD123 in 1965.

E. W. Dijkstra. 1972a. The humble programmer. *Communications of the ACM*, 15(10): 859–866. DOI: 10.1145/355604.361591.

E. W. Dijkstra. 1972b. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds., *Structured programming*, pp. 1–82. Academic Press Ltd. originally published as EWD249 in 1969.

E. W. Dijkstra. 1974a. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11): 643–644. DOI: 10.1145/361179.361202. originally published as EWD426 in 1974.

E. W. Dijkstra. 1974b. Programming as a discipline of mathematical nature. *The American Mathematical Monthly*, 81(6): 608–612. DOI: 10.1080/00029890.1974.11993624. originally published as EWD361 in 1973.

E. W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18: 453–457. DOI: 10.1145/360933.360975.

E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J.

E. W. Dijkstra, 1978a. A review of the 1977 Turing Award Lecture by John Backus. EWD692. Available at http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD692.PDF.

E. W. Dijkstra. 1978b. A political pamphlet from the Middle Ages. *ACM SIGSOFT Software Engineering Notes*, 3: 14–16. DOI: 10.1145/1005888.1005890. originally published as EWD638 in 1977.

E. W. Dijkstra. 1982a. How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective*, pp. 129–131. Springer-Verlag. originally published as EWD498 in 1975.

E. W. Dijkstra. 1982b. A "non trip report" from E.W. Dijkstra. In *Selected Writings on Computing: A Personal Perspective*, pp. 200–204. Springer-Verlag. originally published as EWD561 in 1976.

E. W. Dijkstra. 1982c. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pp. 60–66. Springer-Verlag. originally published as EWD447 in 1974.

E. W. Dijkstra, Sept. 1984. Trip report E.W. Dijkstra, Marktoberdorf, 30 July - 12 Aug. 1984. EWD895. Available at http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD895.PDF.

E. W. Dijkstra, Jan. 1987. Twenty-eight years. EWD1000. Available at http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1000.PDF.

E. W. Dijkstra, Dec. 1988a. On the cruelty of really teaching computing science. EWD1036. Available at http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF.

E. W. Dijkstra, Feb. 1988b. A computing scientist's approach to a once-deep theorem of Sylvester's. EWD1016. Available at http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1016.PDF.

E. W. Dijkstra. 1990. Making a fair roulette from a possibly biased coin. *Information Processing Letters*, 36(4): 193. DOI: 10.1016/0020-0190(90)90072-6. originally published as EWD1071 in 1989.

E. W. Dijkstra, Sept. 1992. The knight's tour. EWD1135. Available at https://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1135.PDF.

E. W. Dijkstra, Nov. 1993. "From my Life". EWD1166. Available at http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1166.PDF.

E. W. Dijkstra. 1995. Heuristics for a calculational proof. *Information Processing Letters*, 53(3): 141–143. DOI: 10.1016/0020-0190(94)00196-6. originally published as EWD1174a in 1994.

E. W. Dijkstra, 1995/1996. A somewhat open letter to David Gries. EWD1227. Available at https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1227.PDF.

E. W. Dijkstra, Jan. 1997. Pruning the search tree. EWD1255. Available at https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1255.PDF.

E. W. Dijkstra, 1999. Computing science: Achievements and challenges. EWD1284. Available at http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1284.PDF.

E. W. Dijkstra, Nov. 2000. Answers to questions from students of Software Engineering. EWD1305. Available at https://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1305.PDF.

E. W. Dijkstra. 2002a. EWD1300: the notational conventions I adopted, and why. *Formal Aspects of Computing*, 14(2): 99–107. DOI: 10.1007/s001650200030. originally published as EWD1300 in 2000.

E. W. Dijkstra. 2002b. EWD 1308: What led to "Notes on Structured Programming". In M. Broy and E. Denert, eds., *Software Pioneers*, pp. 340–346. Springer. https://doi.org/10.1007/978-3-642-59412-0_19. DOI: 10.1007/978-3-642-59412-0_19. originally published as EWD1308 in 2001.

E. W. Dijkstra, 2002c. Coxeter's rabbit. EWD1318. Available at https://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1318.PDF.

E. W. Dijkstra. 2009. On the theorem of Pythagoras. *Nieuw Archief voor Wiskunde*, 5(10): 95–99. originally published as EWD975 in 1986.

E. W. Dijkstra, n.d. A book review. EWD669. A review of *Automated Theorem Proving* by Donald W. Loveland. Available at http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD669.PDF.

E. W. Dijkstra and W. H. J. Feijen. 1988. *A Method of Programming*. Addison-Wesley. originally published in Dutch as *Een methode van programmeren*, Academic Service, 1984.

E. W. Dijkstra and C. S. Scholten. 1980. Termination detection for diffusing computations. *Information Processing Letters*, 11(1): 1–4. DOI: 10.1016/0020-0190(80)90021-6. originally published as EWD687 in 1978.

E. W. Dijkstra and C. S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag, New York.

E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. 1983. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5): 217–219. DOI: doi:10.1016/0020-0190(83)90092-3. originally published as EWD840 in 1982.

S. Dolev. 2000. *Self-Stabilization*. MIT Press, Cambridge, MA.

R. W. Floyd. 1967. Nondeterministic algorithms. *J. ACM*, 14(4): 636–644. DOI: 10.1145/321420.321422.

C. A. Hoare. 1978. Communicating sequential processes. *Communications of the ACM*, 21: 666–677. DOI: 10.1145/359576.359585.

E. S. Loomis. 1940. *The Pythagorean Proposition*, second. National Council of Teachers of Mathematics. Available at http://www.eric.ed.gov/PDFS/ED037335.pdf.

J. McCarthy. 1963. A basis for a mathematical theory of computation. In B. Bradfort and D. Hirschberg, eds., *Computer Programming and Formal Systems*, pp. 33–70. North-Holland.

Memoriam, 2002. In Memoriam Edsger Wybe Dijkstra (1930–2002). Memorial Resolution prepared by a committee, including Jayadev Misra (chair) and Hamilton Richards. Available at https://www.cs.utexas.edu/users/EWD/MemRes(A4).pdf.

M. Raynal. 1986. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, MA.

G. M. C. J. T. G. van den Hove d'Ertsenryck. 2019. *New Insights from Old Programs. The Structure of The First ALGOL 60 System*. PhD thesis, Department of Computer Science, University of Amsterdam, the Netherlands.

M. H. van Emden, 2008. I remember Edsger Dijkstra (1930 – 2002). Available at https://vanemden.wordpress.com/2008/05/06/i-remember-edsger-dijkstra-1930-2002/.

J. von Neumann. 1951. Various techniques used in connection with random digits. *Journal of Research of National Bureau of Standards Applied Math. Series*, 12: 36–38. Available at https://mcnp.lanl.gov/pdf_files/nbs_vonneumann.pdf.

23

Memories of Edsger W. Dijkstra

E. Allen Emerson

Edsger Dijkstra was profoundly influential to all of CS. In my opinion, he was truly the Founder of the Science of Computer Programming. He also made foundational and lasting contributions to programming language design, operating systems, the systematic development of algorithms, and many other subfields of CS.

I also find that the Dijkstra Prize in Distributed Computing is not representative of the scope and lasting impact of his contributions. The ACM Infosys Prize, which has already been renamed the ACM Prize, should be renamed again to become the ACM Dijkstra Prize in Computing. The qualification "Computing" is an approving nod to his view that our discipline, CS, should stand for Computing Science and not Computer Science.

Edsger played an important role in my life. He was not only extremely impactful on my work and my general way of thinking, but a good and ultimately close friend, as well.

23.1 Weakest Preconditions

Most of my research focused on model checking and automata, topics to which Edsger did not contribute. Therefore, one might be tempted to conclude that his work had no impact on mine. But nothing could be farther from the truth.

The very first line of my first publication [Emerson and Clarke 1980] references Edsger's 1976 book *A Discipline of Programming*. In this joint work with my former Ph.D. advisor Ed Clarke, we described various correctness properties of parallel programs using a language involving fixpoints that referred to computation trees that can be unwound into the model generating the paths. We were motivated by Edsger's weakest precondition predicate transformers that for the case of the guarded repetitive command derived its meaning directly from the program text using a fixpoint construction. In the paper, we introduced the computation tree formulae language, CTF in short, and made a remark that it can be viewed as a modal language. We also introduced a language FPF of fixpoint formulae that included weakest preconditions for primitive actions and showed that the CTF formulae can be effectively translated into the FPF formulae.

This work set the foundation for *model checking*, the essence of which lies in evaluating appropriate modal formulas over a finite state graph (the model generating the paths). It was soon followed by our paper [Emerson and Clarke 1982] in which this modal language was explicitly defined and used for *automatic program synthesis* of concurrent programs from high-level branching time temporal logic specifications.

Looking back, one can see that Edsger's weakest precondition transformer *wp* and his weakest liberal precondition transformer *wlp*, in which one does not insist on termination of all computation paths, correspond respectively to the modal operators *AF* and *AG* in our modal language. However, Edsger only thought in terms of *defining* these temporal modalities as extremal fixpoints rather than algorithmically calculating them over the finite state graph induced by the program.

23.2 Fairness

Around 1985, I became interested in reasoning about *fairness*, a property intrinsically relevant in the study of nondeterministic and concurrent programs. My motivation was to incorporate it into model checking so that one could deal with automatic verification of concurrent programs under various fairness assumptions. Initial results obtained jointly with my first Ph.D. student, Chin-Laung Lei, were published in the proceedings of two conferences in 1985; the more comprehensive version appeared as a journal paper [Emerson and Lei 1987].

Much later, I was surprised to learn that Edsger had introduced fairness, both the concept and the term, already in 1971 in his EWD310 "Hierarchical ordening of sequential processes" that appeared the same year in the first issue of *Acta Informatica* as Dijkstra [1971] (with "ordening" changed to "ordering"). Edsger discovered fairness from first principles. His reasoning was well-motivated by practical considerations. On pages 6 and 7 of EWD310 (p. 120 of Dijkstra [1971]), he stated:

> [...] given the bottom layer, what will be known about the speed ratios with which the different sequential processes progress? Again we have made the most modest assumption we could think of, viz. that they would proceed with speed ratios, unknown but for the fact that the speed ratios would differ from zero. I.e. each process (when logically allowed to proceed, see below)

> is guaranteed to proceed with some unknown, but finite speed. In actual fact we can say more about the way in which the bottom layer grants processor time to the various candidates: it does it "fairly" in the sense that in the long run a number of identical processes will proceed at the same macroscopic speed. But we don't tell, how "long" this run is and the said fairness has hardly a logical function.
>
> This assumption about the relative speeds is a very "thin" one, but as such it has great advantages. From the point of view of the bottom layer, we remark that it is easy to implement: to prevent a running program to monopolize the processor an interrupting clock is all that is necessary. From the point of view of the structure built on top of it is also extremely attractive: the absence of any knowledge about speed ratios forces the designer to code all synchronization measures explicitly. When he has done so he has made a system that is very robust in more than one sense.

Let me now paraphrase these remarks. Assume (as Edsger did) a finite number of processes $1, \ldots, n$. Let us consider a nontrivial enabling condition for process i that we shall denote by *en-i*. Edsger said in effect the following:

> it is always the case that if the enabling condition *en-i* for process i, holds now then eventually, after some finite but unspecified (and indeed unknown) time, process i will be executed, denoted by *ex-i*.

In Linear Temporal Logic, called LTL for short, this statement about process i is captured by $G(\textit{en-i} \longrightarrow F\ \textit{ex-i})$. But this is equivalent to so-called *strong fairness*, represented most simply by the LTL expression $GF\ \textit{en-i} \longrightarrow GF\ \textit{ex-i}$.

For simplicity, let us now assume that there are only two processes, named 1 and 2, and that *en-i*, the process i enabling condition, is *true*. In LTL one would simply write *GF ex-1* and *GF ex-2*.

Then all possible executions can be captured by the extended regular expression

$$(1 + 2)^{\omega},$$

where ω stands for an infinite repetition. In contrast, the assumption that each process is to proceed after some finite time is captured by the extended regular expression

$$(12^{*})^{\omega} \cap (21^{*})^{\omega}.$$

In other words, only processes 1 and 2 would execute and both of them would execute infinitely often.

This notion of fairness, so-called unconditional fairness, involves the notion of “infinitely often,” used in automata theory, namely, the theory of finite state automata running on infinite objects, such as strings or trees.

Thus, Edsger defined both of the most common types of fairness: strong fairness and (implicitly) unconditional fairness.

23.3 Personal Reminiscences

From 1984 until his retirement in 1999, Edsger was a member of the Department of CS at the University of Texas at Austin. I would like to share with the readers some of my recollections of him.

Soon after his arrival, Edsger set up a weekly seminar that he called Austin Tuesday Afternoon Club (ATAC). It was attended by about ten researchers, all personally invited by Edsger. It provided a forum for presenting and discussing our new and recent work.

In 1984, we studied a preprint of my POPL85 paper with my Ph.D. student Chin-Laung Lei. That, as I recall, was my first interaction with Edsger, and I had some trepidation. As we went over the paper Edsger didn’t have very much to say. But by the next week Edsger had time to think it over, and he took extreme umbrage to my prior presentation. The gist of his objections was that the writing was much more characteristic of a salesman than of a scientist. I found his criticism highly distressing, if not downright devastating.

Even so, I returned the next week with an at least partially successful rebuttal of Edsger’s criticisms. I explained that, in my opinion, it was the responsibility of the author(s) to motivate the work by describing the history of issues at hand, what was new in the paper, and perhaps to mention some of the key ideas in the solution. Edsger didn’t entirely agree with my explanations, but he did at least agree that there was some merit to them.

More to the point, I believe that he respected the fact that I had returned to ATAC to give my rebuttal after such withering criticisms the week before. I would identify this brief period as the beginning of our lifelong friendship.

As time went on, I noticed that Edsger did not hesitate to act once he trusted somebody’s judgments. I had the honor of experiencing this when I asked him in 1994 to write a recommendation letter for the ACM Turing award for Amir Pnueli, who indeed got it in 1996. I explained to Edsger that Amir Pnueli was the individual I most admire and respect professionally. That was enough for him. We spent 3 hours going over Amir’s vita. Edsger was going to show me his final letter. Unfortunately, I was off to a conference in Banff and never saw it.

In Austin, Edsger and Ria had bought a house with a deck in back. One day I went over to their house. Edsger was on the deck; on its floor there were five pieces of paper in Edsger's well-known printing. Edsger said that was his two-days output. He also revealed that he would sometimes stop and wait, say, 15 minutes just to choose the right word. I say his composition style was Mozartian because it was one pass. This is to be contrasted with my own composition style that involves many drafts and revisions of earlier drafts. I call it Beethovian.

Due to Edsger's influence, it became fashionable in our Department at UT Austin to use a Montblanc fountain pen. I decided I wanted something different, so I bought a Waterman ballpoint pen, but it skipped![1] Edsger had me over to show me his collection of fine Parker ballpoint pens. Alas, they skipped too! Nonetheless, I took that Waterman pen back.

Edsger was my friend and in spite of the age difference we felt very much at ease with each other. It got to the point where he and Ria would come over about every week for his serving of Chivas Regal. Edsger bought me once a Stetson cowboy hat. On another occasion, he and Ria bought my wife Leisa and me a pair of raw wood rocking chairs, made in Bandera, Texas. Those rockers (painted by Leisa since) are still on our front porch.

Edsger used to talk about somewhat subtle (to me) differences between Americans and Europeans. In his view, Americans were obsessed with money and always wanted to know how much a particular item cost: a point he made to a number of his colleagues. He was of the opinion that one should have some money but should not be too concerned about it.

He and Ria toured the US in their Volkswagen camper dubbed by Edsger the Touring Machine. Edsger used to marvel that the US was such a vast country, but all one culture. He was especially fond of camping out in Palo Duro Canyon in Texas, which is the second largest canyon in the US, after the Grand Canyon.

Edsger was very fond of classical music and had many, many cassettes with this genre of music. He used to take some of them with him in his Touring Machine, and in particular listened to them in Palo Duro Canyon. I asked him once: "Could he compose?" He replied: "Sadly, no." He had tried it but without success. Around 2001, after Edsger had returned for good to Nuenen in the Netherlands, I would call him on occasion. Each time I could hear classical music playing in the background.

Leisa and I dined once, in summer 1987, at his and Ria's home in Nuenen. I recall that we were served white asparagus and eggs. After dinner they showed us a house where Vincent van Gogh had once lived. It was just a couple of streets over.

Fifteen years later, in August 2002, I was again in Nuenen, this time for the sad reason to attend Edsger's funeral for which I flew from Austin. Tony Hoare and my Austin colleague J. Moore came straight from Marktoberdorf to Nuenen to speak

[1] A ballpoint pen "skips" when a drawn line has a blank interior region.

at Edsger's service. Almost twenty years have passed since, but Edsger remains in my mind.

References

E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inform.* 1, 115–138. DOI: https://doi.org/10.1007/BF00289519. Reprinted in P. Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming*. Springer, 198–227. DOI: https://doi.org/10.1007/978-1-4757-3472-0_5.

E. A. Emerson and E. M. Clarke. 1980. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and J. van Leeuwen (Eds.), *Proceedings Automata, Languages and Programming, 7th Colloquium*. Vol. 85: Lecture Notes in Computer Science. Springer, 169–181. DOI: https://doi.org/10.1007/3-540-10003-2_69.

E. A. Emerson and E. M. Clarke. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.* 2, 3, 241–266. DOI: https://doi.org/10.1016/0167-6423(83)90017-5.

E. A. Emerson and C.-L. Lei. 1987. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.* 8, 3, 275–306. DOI: https://doi.org/10.1016/0167-6423(87)90036-0.

24

Reflections on Edsger and His Influence

David Gries

In summer 1971, by chance, Edsger and I taught week-long courses in Maryland at the same time. I taught compiler construction using my new textbook [Gries 1971]; Edsger taught operating systems. We sat in each other's lectures. We discussed my text on compiler construction, parts of which he took issue with. At the time, I didn't know that he had written the first Algol 60 compiler! Perhaps that was a blessing, for if I had known all that he had done, talking about compiler construction in front of him might have been difficult.

In fact, he and I had attended the first NATO Conference on Software Engineering [Naur and Randell 1969], but, strange as it may seem, we didn't notice each other. I had received my Ph.D. in math in June 1966 from the Munich Institute of Technology, just two years earlier, and I had been invited to the NATO Conference by my Ph.D. advisor, Fritz Bauer, as a sort of high-level gofer. I didn't know many of the people at the Conference.

Our wives, Elaine and Ria, were with us in Maryland, and we got along well. Elaine and I taught them how to throw a frisbee, which had just become popular. Thus began a friendship that lasted until his death.

Edsger —[1]and Tony Hoare— profoundly influenced me, and I soon switched my research from compilers to programming methodology and related topics. I hesitate to think what a flop my career might have been if I hadn't met these giants.

Edsger visited Cornell fairly often after that, and his daughter, Femke, studied here for a year. To get a taste for what Edsger was like, read his report of a 1980 trip to Cornell (see EWD727 [Dijkstra 1980]). In a way that no other computing scientist could do, Edsger weaved a journal of his trip with honest opinions of the places

[1] As in Chapter 7, in deference to Dijkstra's use of em dashes, a space is used before and after each parenthetical remark delimited by em dashes.

and people he visited, a tongue-in-cheek list of "language rules," and his view of the problems faced by CS in the US. After reading this EWD, you will want to read more.

I didn't feel it then, but I now view the 1970s and early 1980s as a magical time. The Marktoberdorf Summer Schools were high points. About ten faculty would lecture to 100 Ph.D. students from around the world, mostly on programming methodology and related topics. The lectures themselves, the discussions after lectures, the camaraderie among faculty and students, the social events, the banquet — everything contributed to a wonderful time— both educationally and socially. Also, the annual meetings of IFIP Working Group 2.3 on Programming Methodology were quite different from normal conferences. People talked less about what they had done and more about what they were doing, and they asked for and received advice and suggestions. The advice I got at various times was very helpful.

Edsger had a great deal to do with how the Summer Schools and especially the WG2.3 meetings were run. This quote from EWD714 [Dijkstra 1979], —a must read— shows you what Edsger thought of WG2.3 and introduces you to his scientific mission.

> "[We had] the shared recognition that . . . only the most effective application of mathematical method . . . could be hoped to solve the problems adequately. The charter was clear: searching for relevant abstractions and separation of concerns that are specific to programming and learning how to avoid the explosion of formulae when dealing with stuff more complicated than mathematicians had ever formally dealt with before. I found this charter sufficiently challenging to devote my scientific life to it; at the same time this charter was sufficiently unpopular "in the real-world" . . . to justify the protection of an IFIP Working Group . . . "

Edsger, Tony, and I also held weeklong schools in the 1980s. This started when Juris Reinfelds of the University of Wollongong invited us to hold the "First Wollongong Summer School on The Science of Programming." It took place at Perisher Valley, Australia, in early 1983. Netty van Gasteren, one of Edsger's Ph.D. students, came as a Teaching Assistant, and Edsger and Tony's wives, Ria and Jill, also came. We had lectured together before, in Santa Cruz in 1979 (EWD714 [Dijkstra 1979]), so this was not really a new thing. But traveling through Australia with the Dijkstras and Hoares, discussing what we would say in our lectures, and listening to Edsger and Tony lecture was not only fun but educational for me as well. To see Edsger's thoughts of this trip, read his trip report EWD847 [Dijkstra 1983].

In June 1985, we gave a "Newport Summer School" in Newport, RI. This time Edsger, Tony, their wives, and Edsger's colleague Wim Feijen first came to Ithaca, giving us a chance to prepare the summer school. The whole Computer Science Department joined in welcoming and talking with Edsger and Tony. After several days, we flew together to Newport. Juris Reinfelds, who had organized the school in Australia, was a participant in the Summer School. He was impressed by the progress we three had made since his school two years earlier. Yes, indeed, the 1980s was a period of rapid advances in our ability to teach the development of program and proof hand-in-hand. Read Edsger's EWD923 [Dijkstra 1985] to see how this trip went and to get a feel for what it meant for me to work with Edsger and Tony.

Looking back, I chide myself for not keeping a diary as I went through my career. I just don't remember everything —who does? But, in a way, Edsger's trip reports fills that gap for me. Reading them brings back many fond memories.

In my opinion, three characteristics helped make Edsger *the* dominating figure in computing. First was his brilliant mind. Who else could develop the shortest-path algorithm in their head in 20 minutes while having coffee with their fiancé? But who else would then recognize that "brainpower is by far our scarcest resource?" (From his Turing Award speech, EWD340, *The Humble Programmer* [Dijkstra 1972]).

Edsger spent his career thinking about method, in programming and mathematics. Tony Hoare provided tools for proving programs correct, with his axiomatic basis for computer programming. But Edsger showed us how to *develop* programs using these tools. Edsger also spent his scientific life contributing to mathematical method as no one else has done. It was enlightening at a Marktoberdorf Summer School to see him present calculational logic, which led to principles and strategies for developing formal proofs. But I am much more awed reading his EWDs, which are filled with developments of arguments and proofs, discussions, developments of proofs and programs, always with an aim of illuminating *method*.

Second, Edsger had a passion for computing and its growing importance. Here's what he said in his Turing Award speech ([Dijkstra 1972]) about IBM 360 computers in the mid 1960s: ". . . the design embodied such serious flaws that I felt that with a single stroke the progress of computing science had been retarded by at least ten years; it was then that I had the blackest week in the whole of my professional life." This was based on his knowledge of the problems of concurrency and interrupts, for he had helped develop both a computer and its operating system in the 1950s and 1960s.

Third, Edsger wrote and spoke with honesty and candor. Gerry Salton of Cornell said about one of his trip reports, "Dijkstra's right, but we don't say such things." Edsger would have replied, "If you don't say such things, how can we hope to improve?"

Here's what happened to me at a Marktoberdorf Summer School in the 1970s. Writing an assignment like "$x := 5$" on the board, I said "x equals 5." From the back of the room came a loud, booming, Dijkstra voice, "BECOMES." I was startled, but I regained my composure and said, "Thanks. If I make that mistake again, tell me." I made it once more during that lecture. You can envision what happened. I have never made that mistake since then. These days, I have fun recounting this episode when students make the same mistake.

Edsger critiqued not the person but only what they said, and later one could drink a beer and laugh as if nothing happened. Technical differences and shortcomings should be treated this way.

Few people knew that Edsger and Ria were social animals. During our 1989–1990 sabbatical year in Austin, we had dinner at each other's houses, alternating weeks. The evenings would be spent talking, discussing a CS issue, or reading a new chapter of the book *A Logical Approach to Discrete Math*, which I was writing with Fred Schneider and which eventually appeared as [Gries and Schneider 1993]. Sometimes, Edsger would play his piano, a Bösendorfer.

Also, Edsger and Ria would drop in unannounced on an evening. A knock on the door, and in they came. The first time it was a shock, but we soon looked forward to such visits. They visited several people in the Austin CS Department in this manner.

There's so much more I could say about Edsger. The best way to learn more about this giant is to read his EWDs on this website: E.W. Dijkstra Archive [Dijkstra 1930–2002]. Chapter 30, by Hamilton Richards, discusses the creation of this website.

My scientific life was spent working and playing with this giant, and I am ever so grateful that we met.

References

E. W. Dijkstra. 1930–2002. E.W. Dijkstra Archive. The manuscripts of Edsger W. Dijkstra. https://www.cs.utexas.edu/users/EWD/.

E. W. Dijkstra. 1972. The humble programmer. *Commun. ACM* 15, 859–866. DOI: https://doi.org/10.1145/355604.361591.

E. W. Dijkstra. January. 1979. Trip report E.W. Dijkstra, Mission Viejo, Santa Cruz, Austin, 29 July–8 September 1979. EWD714. https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD714. PDF.

E. W. Dijkstra. January. 1980. Trip report E.W. Dijkstra, U.S.A., 12 Jan.–2 Feb. 1980. EWD727. https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD727.PDF.

E. W. Dijkstra. 1983. Trip report E.W. Dijkstra, Australia, 19 Jan. 1983–12 Feb. 1983. EWD847. https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD847.PDF.

E. W. Dijkstra. 1985. Trip report E.W. Dijkstra, Ithaca, Newport, 30 May–13 June 1985. EWD923. https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD923.PDF.

D. Gries. 1971. *Compiler Construction for Digital Computers*. John Wiley and Sons, New York.

D. Gries and F. B. Schneider. 1993. *A Logical Approach to Discrete Math*. Springer Verlag, New York. ISBN: 978-0-387-94115-8. DOI: https://doi.org/10.1007/978-1-4757-3837-7.

P. Naur and B. Randell (Eds.). 1969. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch-Partenkirchen, Germany, 7–11 October. 1968. Scientific Affairs Division, NATO, Brussels. Reprinted in *Software Engineering: Concepts and Techniques* (J. M. Buxton, P. Naur and B. Randell, Eds.), Petrocelli/Charter, New York, 1976. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

25

Forty Years with Edsger

Tony Hoare

My 40 years with Edsger started in April 1961 in Brighton, England. He was then just 30 years old, me nearly 4 years younger. He was lecturing on a course for programming in Algol 60, together with Peter Landin and Peter Naur. The latter had drafted and edited the Algol 60 Report, a complete and very clear definition of the syntax and semantics of that language. It was printed as an A5 booklet of 23 typewritten pages. Yet it contained all the information needed by an implementor of the language and by its user. And both of them could understand it. I can vouch for this by personal experience.

When the class was set to solve exercises, I decided instead to write an implementation of a recursive sorting algorithm that I had discovered the year before but had failed then to find an implementation for. I used recursion, which made it easy. I showed my solution to Peter Landin. He was impressed and beckoned to Peter Naur to have a look. But the third lecturer was standing some distance away, and never saw it on that occasion. He produced later an elegant extension of it (smoothsort), which preserved the sequence of elements with the same primary key.

Although I attended Edsger's lectures, we later both confessed that neither of us could remember the other. I attributed his lapse of memory to the fact that I was only one student among many. He kindly attributed my lapse to the difficulty of distinguishing between two lecturers both wearing beards.

With me on the course was my future wife Jill, who was then my colleague. We both worked as programmers for a small British computer manufacturer, a division of Elliott Automation Ltd. We were implementing an Algol 60 subset to run on the next generation of Elliott computers, which would be 60 times faster than their current machine but with the same architecture. At the same time, Edsger was implementing a much fuller subset of the language for a small Dutch computer manufacturer, Electrologica.

* * *

As a result of our experience in implementing Algol 60, we were both selected as members of IFIP Working Group 2.1. In September 1963, we both attended its second meeting in Delft. This Group was charged with the curation of the Algol 60 language—including corrections to the Report, the design of a subset of the language, and of optional extensions to it. And finally, the design of a language that would replace it.

The subset under discussion at the meeting had been designed by the so-called ALCOR group of numerical analysts in the Group. They removed recursion from Algol 60 on the grounds of its alleged inefficiency. Edsger's answer was that recursion was a useful programming tool, and every workman should be allowed to fall in love with their tools. I certainly had done so. It is clear that Edsger was a man after my own heart.

To my shame, when discussion turned to suggestions for extension of the language, I reported a request of potential customers of my company's new machine. It was to follow the example of FORTRAN II and provide an option for omission of variable declarations together with their types.

Kindly waiting until the next lunch break, Edsger approached me together with Peter Naur. They explained to me quietly just what a bad idea it was. It would be a source of almost undebuggable errors caused by the merest misspelling of an identifier. The only protection against such errors was to build redundancy into the programming language. This would enable the compiler to detect source program errors and report them instead of running the program.

I made such checks the guiding principle for the implementation of the Elliott Algol compiler. Because of its recursive structure, the compiler detected all occurrences of all syntax errors and all type errors and all scope errors. And it produced code that detected all runtime errors, including subscript errors and numeric overflows. These errors could make the program totally unpredictable from inspection of the code, and therefore undebuggable.

This incident started me on a most productive direction for my career of research. I spent ten years exploring the design of types for variables and for structured data. I published some of them immediately in the Algol Bulletin [1959–1988], and I finally assembled them under the title "Notes on data structuring" that eventually appeared as a chapter [Hoare 1972a] in Dahl et al. [1972]. Edsger wrote the first chapter, "Notes on structured programming" [Dijkstra 1972a].

The last meeting of WG2.1 that I attended was the one which made the recommendation of Algol 68 as a successor to Algol 60. It was also Edsger's last attendance. We both abhorred the complexity of the new language, and its manual of 140 journal pages, [van Wijngaarden et al. 1969]. Even the definition of its syntax used a cumbrous new notation, which we found incomprehensible. Others shared our misgivings, and under the leadership of Edsger we wrote a Minority report (reproduced in EWD252 [Dijkstra 1968]), discommending the language. He drafted the first paragraph:

> We regard the current Report on Algorithmic Language ALGOL 68 as the fruit of an effort to apply a methodology for language definition to a newly designed programming language. We regard the effort as an experiment and professional honesty compels us to state that in our considered opinion we judge the experiment to be a failure in both respects.

* * *

The next ten years were our most intensive period of interaction of our researches into the Theory of Programming. The period began with a seminar in 1971, held at the Queen's University, Belfast. The lectures, together with subsequent discussions, were recorded in the book *Operating System Techniques* [Hoare and Perrot 1972]. I gave a talk on conditional critical regions. They take the form:

with <resource variables> **when** <wake-up condition> **do** <critical region>

Other concurrent processes would have to discharge the responsibility for waking up this process by making the wake-up condition true.

Edsger gave the next talk on "Hierarchical ordering of sequential processes," published as [Dijkstra 1972b]. At the end of it, he described his concept of the "Secretary," acting as part of the kernel of a multi-programming system. It consisted essentially of a set of conditional critical regions sharing the same resource. They were local to the secretary, and scope checks would prevent any other process from accessing them. These were called as procedures, possibly with parameters, by the multiprogrammed threads.

The secretary was slightly modified in its scheduling capabilities by Per Brinch Hansen and me working together. We renamed the resulting language feature as a Monitor [Hoare 1974, Brinch Hansen 1974, section 7.2].

Ideas of secretaries or monitors stimulated a new ten-year phase in my research on concurrency, culminating in the publication in 1985 of a textbook *Communicating Sequential Processes* [Hoare 1985].

The following paragraphs examine the interactions of Edsger's discoveries with my design of the language of Communicating Sequential Processes (CSP). They are interesting as a story of how two scientists can start from different conceptual backgrounds (physics and philosophy) and develop from them the same theory. It greatly increases scientific confidence in the validity and wide applicability of both the original theories.

The interaction will be described in greater technical detail, as an extension of the conditional critical region of the monitor concept, supplemented by the ideas expounded in Edsger's seminal 1975 EWD472 "Guarded commands, nondeterminacy and formal derivation of programs," published as Dijkstra [1975]. Edsger initially wrote all his proofs of programs informally and was resistant to the stark formality of Hoare Logic. Grocery, he called it.

But in this article, he embraced formality enthusiastically. It led to his own brilliant calculus of weakest preconditions, soon after used in his book *A Discipline of Programming*. It also introduced the language of guarded commands. In my 1984 paper entitled "Programs are predicates" [Hoare 1984], I suggested that the language can be reduced to a simple set of algebraic axioms, much more elegant than the proof rules of Hoare Logic. The interaction of our ideas was indeed intense.

I took over guarded commands directly into CSP. As I stated in the ACM Interview,[1] I don't think I would have dared to employ such a strange syntax if Edsger hadn't paved the way with his beautiful guarded command language.

I felt it was very important that, if a process attempts to test whether an output is available for further input, it should do so with a command that makes possible that the output would take place simultaneously because I didn't want anybody testing the availability of something and then not using it when it had been found to be available.

At the time, microprocessors were very cheap and fast but the sharing of memory between the microprocessors was expensive and slow. So, I also decided to depart radically from Edsger's concurrency model with shared state because communication fitted very well the architecture of the implementation and could be very fast.

In transferring Edsger's ideas to CSP, the conditional critical region of a monitor plays the role of a single guarded command. The syntax had no selection of required resources, and was abbreviated to

$$<\text{wake-up condition}> \;\rightarrow\; <\text{critical region}>$$

The guarded command set was a collection of guarded commands, connected by a nondeterministic choice operator □. This selects just one of the alternatives of the set. The proof rule for the whole set requires the wake-up condition proof to guarantee the existence of at least one such choice, and each possible choice must satisfy the precondition of the following critical region.

[1] Its transcript can be found in an appendix of Jones and Misra [2021].

The implementor is generally expected to choose the command whose guard is the first to become true as in the external choice of CSP. This scheduling strategy typically halves the expected length of wait and reduces the standard deviation even further. The resulting reduction of unpredictable latency is significant in solving one of the major complaints, both of interactive users of computers and of the designers of cyberphysical systems. I summarized these benefits in the title of a later lecture entitled "Concurrent programs wait faster," [Hoare 2003], where the paradoxical phrase was suggested by Edsger.

Edsger's conceptual breakthrough in 1975 was to realize that whenever two guards were simultaneously true the implementer could make an arbitrary choice. This was called "demonic choice" because the proof of a program rules out all the possibly malicious choices made by the demon. The opposite kind of choice is called "angelic choice," which is postulated by Automata Theorists to simplify their analysis of algorithms. In logic-based languages like Prolog, angelic choice is a requirement on the implementor, who has to use a potentially exhaustive tree search to find a solution of a set of implications expressed in predicate logic.

Formal development was just the replacement of a generally nondeterminate abstract design by a more concrete design that is more deterministic. The opposite direction of design is called abstraction, which is essential to formal development. It is normally implemented by method declarations and method calls, as explained in my 1972 paper "Proof of correctness of data representations" [Hoare 1972b].

In the late seventies my research interactions with Edsger became personal and my meetings then intensified. It included our participation as lecturers and our joint Directorship of the Marktoberdorf Summer Schools, including joint editorships of the proceedings. Edsger was involved as a lecturer, director, and editor of the proceedings from 1972 until 1998 when he retired from the Directorship. I also spent a sabbatical year, visiting him at the University of Texas at Austin (1986–1987).

* * *

Jill and I last met Edsger in his house at Nuenen, in August 2002. He had frequently expressed the view that it was the duty of the dying to comfort those who will survive them. Accordingly, he and Ria invited their foreign friends to visit him for a few days. We were joined at mealtimes by more local friends. He was completely lucid, but too tired to talk about technical matters.

A few weeks later I was lecturing at the Summer School in Marktoberdorf. I had the sad duty of announcing the death of their much-loved Director. I was most comforted by recollection of our long friendship with a man who embodied the best scientific, educational, and personal traditions of the past. Indeed, in his humility about his own intellectual powers and in his interactive method of teaching, he was reminiscent of the ancient Athenian philosopher Socrates. Socrates also comforted his friends as he went to his early death at the hands of the Athenian executioner (Plato, *Apology*).

I was invited to give a brief presentation at his funeral. I cut short my stay in Marktoberdorf to attend it. My speech ended with an assessment that

> ...[he had laid] the foundations that would establish computing as a rigorous scientific discipline; and in his research and in his teaching and in his writing, he would pursue perfection to the exclusion of all other concerns. From these commitments he never deviated, and that is how he has made to his chosen subject of study the greatest contribution that any one person could make in any one lifetime.

* * *

Edsger was a person of many idiosyncrasies: in his hates, his fears, his clothes, his working environment, his work practices, and his priority of values. He had a strong aversion to cheese, both its smell and its taste. With a similar aversion to cucumber, I sympathized. He also had a distaste for broccoli and other foods containing dietary fiber. On taking guests to a standard Texas steak joint, he was pleased to point out that chicken was included as a side dish in the vegetable section of the menu.

He told me that he had once asked his audience whether they believed that Computing was a Science, and why. A bold soul answered "Yes, because computers exist." His rejoinder was "So existence of broccoli proves the existence of Broccoli Science?" His choice here of broccoli as an example was not fortuitous.

He had a strong fear of horses because their behaviour is so unpredictably agitated. I once took him for a walk from my home in Oxford to a nearby meadow. To get there, we needed to take a narrower section of the path, but clearly wide enough for walkers in opposite directions to pass each other. On this section he saw approaching us a man leading a horse. To me he said quietly that he thought we should turn and go back. And so we did.

This fear explains the piquancy of his choice of a metaphor in the discussion of the complexity of programs from his "Notes on structured programming" (EWD249), published as Dijkstra [1972a]:

> Apparently we are too much trained to disregard differences in scale, to treat them as "gradual differences that are not essential" [...]

> Let me give you two examples to rub this in. A one-year old child will crawl on all fours with a speed of, say, one mile per hour. But a speed of a thousand miles per hour is that of a supersonic jet. Considered as objects with moving ability the child and the jet are incomparable, for whatever one can do the other cannot and vice versa. Also: one can close one's eyes and imagine how it feels to be standing in an open place, a prairie or a sea shore, while far away a big, reinless horse is approaching at a gallop, one can "see" it approaching and passing. To do the same with a phalanx of a thousand of these big beasts is mentally impossible: your heart would miss a number of beats by pure panic, if you could!

* * *

Edsger had a strong aversion to certain words and phrases and styles of exposition, which were not uncommon in publications and lectures in the field of Computing Science. He was particularly critical about graphs of a function that had no measurements on their axes and diagrams that had no explanation of the meaning of their lines and arrows. He hated words like "intuitive" and "natural," used as a justification for a concept. He deplored the use of examples as a motive (or even as a substitute) for proper definition. He regarded all these faults as indicative of sloppy thinking.

On finding such a fault in reading a publication, he would take this as a good reason to stop reading at this point. In a lecture he would often interrupt to advise the lecturer how and why to avoid them. In his written trip reports, he often offended his hosts by helping them with the same advice.

He gave long thought to the style and content of each paragraph of his writing. He would not commit it to paper till he knew exactly the complete text. He gave great consideration to the choice of his main working tools and to his apparel. For writing his manuscripts, he chose a Mont Blanc ink pen, which he replaced whenever the gold nib was worn down.

In Austin he decided that the best headdress was a Texas cowboy hat (see Figure 25.1); the best for his collar was a Texas choker, which tightens around the neck without a knot. For his trousers he chose a pair of shorts, and for his feet a pair of sandals instead of knee-length jackboots favored by a true Texan. Instead of pockets he carried a leather man-purse on a strap over his shoulder. It was an incongruous combination, but one copied closely by several of his immediate disciples.

Edsger disliked the projection of slides as an aid to delivering lecture. He was particularly scathing about multicolored slides. For his own lectures to small

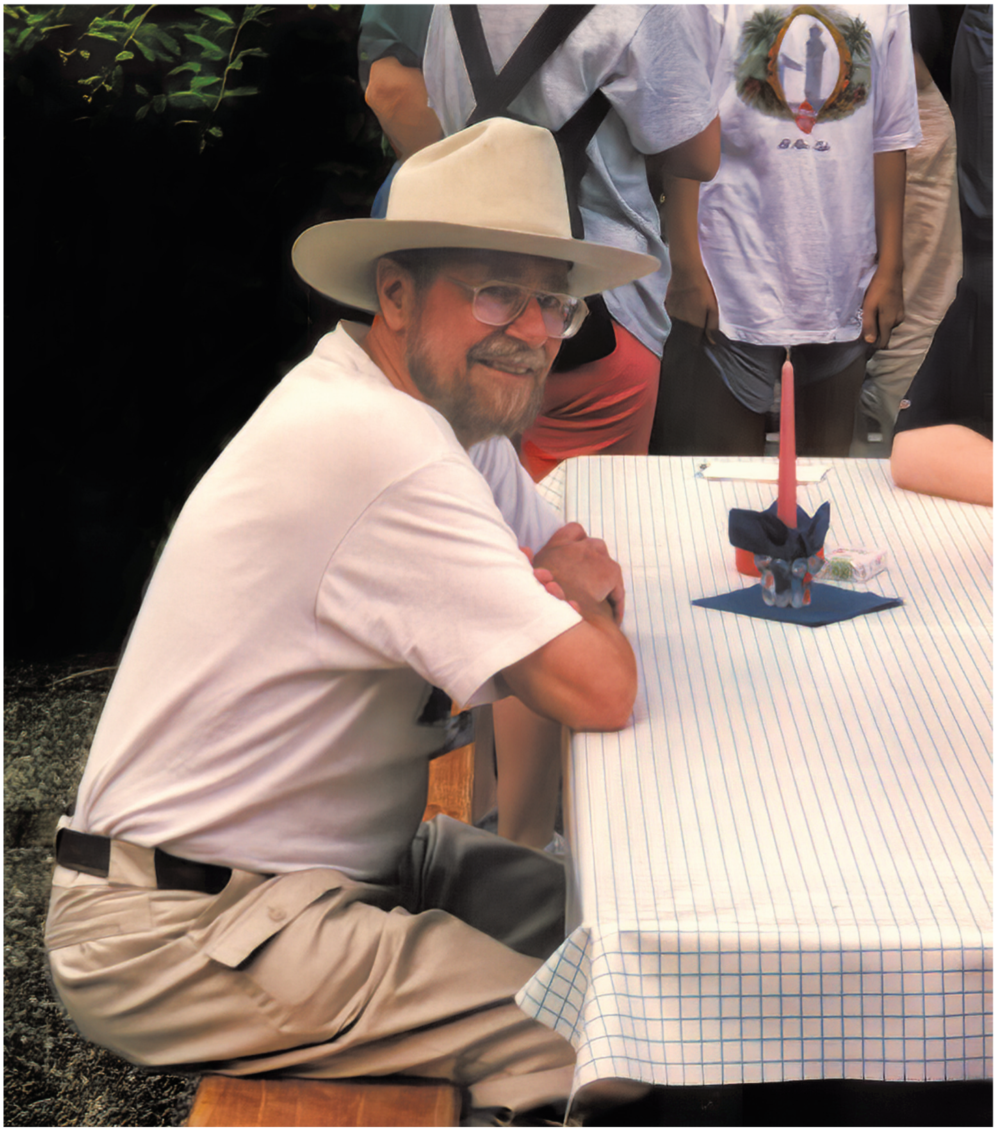


Figure 25.1 Edsger W. Dijkstra wearing a Texas cowboy hat. Photo by Rajeev Joshi taken in Marktoberdorf in 1988. (Photo courtesy of Rajeev Joshi.)

audiences, he used blackboard and chalk, just as in the traditional lessons taught in schools and universities. Otherwise, he used a roll of acetate foil mounted to an overhead projector. Line by line by hand, he wrote his notes on it, before winding the roll on. This is immensely more useful and memorable to the audience than a screenshot, in which the text was often too small to read, even from the first row.

Edsger deplored sales talk in a scientific article. What should appear in a journal is an account of the assembled evidence for its theory, together with a full declaration of any deficiencies and any grounds for doubt. This leaves it to the reader to evaluate the proposed conclusions independently.

I shared his distaste for operational semantics. An axiomatic semantics can explain (maybe indirectly) the purpose of a concept and how it can be properly used (e.g., a chair is for sitting in). An operational semantics would have to give instructions for building a chair. In fact, he objected even to the term "Computer Science." He likened this to naming Astronomy as "Telescope Science," defining a science by its instruments rather than by the area of its search for truth.

He never gave any attention to program testing. He claimed that none of the programs developed and proved in his books and articles had ever been tested in advance of publication. For several years of my academic career, I used to follow the same doctrine about testing. But eventually I came to believe we were both wrong. In Hoare [1996], I extensively argued that in all other branches of engineering proof and testing have complementary roles, and both contribute confidence in the safety and serviceability of a delivered product. Indeed, assertions sprinkled throughout a program can give relevant help, both in locating and explaining errors made during a test of a program and in proving that tests are not needed. In the past, this has been the best way of encouraging programmers to use assertions.

Now there are more persuasive arguments. Increasingly, programs are applied in environments where even a single error in a running program is one error too many. For such applications, only proof can give the requisite assurance. Increasingly, these proofs can be checked, or even generated, by automatic proof tools. So, there is a hope that proofs of the most important properties of the most important computer programs will eventually be more reliable than many existing published proofs of mathematical theorems.

Edsger often encountered the objection that his programs and his ideas do not scale to the problems of the real world. His rejoinder was to define such problems merely as those that remain after you have ignored their known solutions. I would now give a longer reply. The primary teaching of Edsger about structures and about abstractions in programming are eminently scalable. Abstraction from the real world, like the theories in the natural sciences, makes a natural law exponentially shorter than the experimental data that affirms it. Similarly, specification of a program can be exponentially shorter than the program itself, which is similarly shorter than the logs of its execution. Structure in programs permits design of large programs to proceed stepwise, according to correctness-preserving transformations, thus ensuring that they are correct by construction.

Furthermore, the correctness of these transformations can be checked by automated proof tools running on the computer itself. Independently of reliability, Edsger and I agree with pure mathematicians, who appreciate the breathtaking beauty of the ideas behind a short proof.

Most highly, he valued simplicity. He used to be a heavy smoker. He gave up overnight after a visitor described the complicated algorithm he had recently used to quit smoking. He thought "Surely it can't be that complicated" and proved it by quitting immediately, with no subsequent relapse.

Edsger greatly valued brevity. He quoted Confucius as offering an apology for writing such a long letter, because he had no time to shorten it. He frequently attributed to Confucius the precept that a picture is worth a thousand words. He just made a different claim, namely that "A formula is worth a thousand pictures." Sometimes he followed the contrapositive of Confucius's precept and drew highly illuminating pictures; they were all worth more than a thousand words.

He had no interest in economics, and he described as inappropriate the seemingly frequent use of the Almighty Dollar as a unit of measurement for thought. He believed that the only way of "making money" was to take it from someone else's pocket. I disagreed. In those days of exponential increase in computer speed, memory, accompanied by decrease in cost, it seemed acceptable to me to make money from the accruing benefit, leaving much more benefit for the customer. Now in less prosperous times, I agree with Edsger's view. The overall accumulated wealth of the rich at any time is exactly matched by the overall accumulated debt of the poor.

References

Algol Bulletin. 1959–1988. https://archive.computerhistory.org/resources/text/algol/algol_bulletin/. Produced by the IFIP Working Group 2.1 on ALGOL from March 1959 to August 1988.

P. Brinch Hansen. 1974. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ.

O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.). 1972. *Structured Programming*. Academic Press.

E. W. Dijkstra, 1968. Mijn laatste verslag van een bijeenkomst van W.G.2.1. EWD252. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD252.PDF.

E. W. Dijkstra. 1972a. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.), *Structured Programming*. Academic Press, 1–82.

E. W. Dijkstra. 1972b. Hierarchical ordering of sequential processes. In C. A. R. Hoare and R. H. Perrot (Eds.), *Operating Systems Techniques*. Academic Press, 72–93.

E. W. Dijkstra. 1975. Guarded commands, non-determinacy and formal derivation of programs. *Commun. ACM* 18, 453–457. DOI: https://doi.org/10.1145/360933.360975.

C. A. R. Hoare. 1972a. Notes on data structuring. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.), *Structured Programming*. Academic Press, 83–174.

C. A. R. Hoare. 1972b. Proof of correctness of data representations. *Acta Inf.* 1, 271–281. DOI: https://doi.org/10.1007/BF00289507.

C. A. R. Hoare. 1974. Monitors: An operating system structuring concept. *Commun. ACM* 17, 10, 549–557. DOI: https://doi.org/10.1145/355620.361161.

C. A. R. Hoare. 1984. Programs are predicates. *Philos. Trans. R. Soc. London. A Math. Phys. Eng. Sci.* 312, 1522, 475–489. DOI: https://doi.org/10.1098/rsta.1984.0071.

C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall. ISBN 0-13-153271-5.

C. A. R. Hoare. 1996. How did software get so reliable without proof? In *Proceedings of FME'96: Industrial Benefit and Advances in Formal Methods*, Vol. 1051: Lecture Notes in Computer Science. Springer, 1–17. DOI: https://doi.org/10.1007/3-540-60973-3_77.

C. A. R. Hoare, 2003. Concurrent programs wait faster. Slides of a lecture delivered at Microsoft Research Tech Fest 2003. https://www.powershow.com/view/17ffdf-YzRhN/Concurrent_programs_wait_faster_powerpoint_ppt_presentation.

C. A. R. Hoare and R. H. Perrot (Eds.). 1972. *Operating Systems Techniques*. Academic Press Ltd.

C. B. Jones and J. Misra (Eds.). 2021. *Theories of Programming: The Life and Works of Tony Hoare*. ACM/Morgan & Claypool. DOI: https://doi.org/10.1145/3477355.

A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. 1969. Report on the algorithmic language ALGOL 68. *Numer. Math.* 14, 2, 79–218.

26

Edsger Dijkstra—Some Reminiscences

Brian Randell

26.1 Introduction

It is a pleasure and an honor to contribute some reminiscences about my interactions over the years with the late great Edsger Dijkstra. In doing so I am employing both my memory and earlier attempts I have made at documenting and expressing my gratitude for these interactions, in particular an invited talk at the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003) [Randell 2003], the text of which I have drawn on extensively.[1]

26.2 Brighton 1961

For me, the story of Edsger Dijkstra begins in April 1961 when I attended a lecture that he gave on his Algol 60 Compiler at a conference in Brighton. (I was approaching my 25th birthday, he was already 30.) The Algol 60 Report [Backus et al. 1960], famous both for the Algol language it defined and as an example of how to define a programming language, in particular its syntax, documented the work of the Algol Committee, but was itself largely the work of Peter Naur. Dijkstra was at this time a programmer at the Mathematical Centre, Amsterdam, whose Director, Aad van Wijngaarden, was a member of the Algol Committee.

Within seven months of the publication of the Algol 60 Report, Edsger Dijkstra and a colleague, Jaap Zonneveld, had completed a compiler for virtually all of Algol 60, despite the fact that the language had many highly novel features and great generality. (For example, it had recursive procedures, hierarchical name scoping, multidimensional arrays whose size could depend on input calculations, powerful parameter passing mechanisms, etc. These were facilities that far exceeded in

[1] Extracts from Randell [2003] are incorporated in the present text by kind permission of the IEEE.

generality and elegance those of the two main languages then in use, Fortran and Cobol.) Indeed, Dijkstra and his colleague produced their compiler at a time when no one on the committee had yet figured out how to compile the language whose facilities they had just approved.

At this time, I was working for the Atomic Power Division of the English Electric Company in its large development laboratory at Whetstone, a village outside the East Midlands city of Leicester. I was managing a small "Automatic Programming Section," which had been established after I and a colleague, Mike Kelly, had—as a bootleg project—produced a compiler for English Electric's first computer, the DEUCE.

The first DEUCE was produced in 1955, but its design was very closely based on Alan Turing's original plans for the ACE computer at the UK's National Physical Laboratory [Turing 1945]. Hence its order code was extremely cleverly designed but usable really well only by extremely clever programmers, that is, by the likes of Alan Turing himself—and Mike Kelly. Hence, Mike and I decided there was a need for a somewhat higher-level language that could be used by ordinary mortals, even though our assigned task was to produce nuclear application programs. The result was a programming system called EASICODE, which exploited an obscure feature of DEUCE's instruction set that was discovered by Mike, in order to implement a very efficient low level-level interpreter. The first major use of EASICODE was for an engineering application and was spectacularly successful. Deployment of the completed application saved more DEUCE computer time than the total time Mike and I had spent (surreptitiously) on the development of the compiler.

Despite Mike leaving to join IBM at Hursley, it was decided that work on programming aids was worthy of continuation at Whetstone. An Automatic Programming Section was created, and I was joined by a new colleague, Lawford Russell. At the time of the Brighton Conference, he and I were then planning—for an as yet unspecified language—a compiler aimed at easing the task of developing programs for English Electric's planned new computer, the KDF9 [Davis 1960]. The KDF9 was a very innovative computer incorporating a small arithmetic push-down store, a subroutine "nesting store," and much else. After Dijkstra's talk, someone suggested that Lawford and I should target our work on Algol and seek Dijkstra's assistance. This led—eventually, since it took quite a while to get permission—to our first ever expenses-paid foreign travel.

26.3 Visit to the Mathematical Centre

Lawford and I spent a wonderful, but highly intensive, week in Amsterdam—learning from Dijkstra about the Electrologica X1 compiler—and discussing plans for a compiler for KDF9, a compiler that later became known as the Whetstone Algol Compiler. What we aimed at was a fast compiler for use during program development, there being plans elsewhere in English Electric for a sophisticated optimizing compiler for the machine. Hence, we decided that we would produce a single-pass compiler that would convert Algol into an intermediate language, in fact a precursor of P-code [Kristensen et al. 1974]. In effect, we designed a computer architecture that would be convenient to compile for and that would be implemented by an interpreter that provided extensive program testing facilities.

Many details, small and large, of that wonderful week are burned into my memory. For example, I recall that the Mathematical Centre carefully booked us into the Hotel Krasnapolsky—they knew from experience that it was virtually impossible for a foreign visitor to Amsterdam to mispronounce the name so badly as to confuse a taxi driver.

I assume we met van Wijngaarden and Zonneveld, but I have no memory of doing so. I recall being told, perhaps during this visit, the story of van Wijngaarden's secretary very worriedly reporting to him that she had just seen Dijkstra and Zonneveld fighting in the corridor—only to be reassured that this was just one of their typically intense technical discussions. And it may be during that week that I learned that Dijkstra and Zonneveld had both grown beards in order to distinguish themselves visually from van Wijngaarden, in order to emphasize that, contrary to his occasional remarks, it was just they who were actually working on the Algol compiler.

Lawford and I spent each morning and afternoon in intensive discussions just with Dijkstra, but he apologetically asked us to look after ourselves at lunchtime, so that he could have a rest from speaking English, though in fact his command of the language was already impressive. Indeed, during a very pleasant evening with Edsger and his wife Ria in their apartment, I recall him seeking linguistic help only with some of the more abstruse vocabulary of a song by Tom Lehrer.

26.4 Our Trip Report and Book

After returning from Amsterdam, we produced a detailed trip report documenting these discussions [Randell and Russell 1962]. Our report concentrated on providing an initial planned design of the interpreted "object program" that KDF9 Algol programs would be compiled into and describing how our planned compiler for this object program might work. To our pride (and now to my regret), Dijkstra used the availability of our report both to fend off further would-be visitors and as an excuse to avoid, or at any rate delay, producing a detailed account of his own Algol 60 compiler.

I've referred to him above as Edsger, but in fact during all this time, and in the correspondence that followed our visit, we were still addressing each other formally. (This was Europe, and the 1960s.)

But then Lawford and I found a way of usefully improving on the strategy employed in Dijkstra's compiler, which he had designed simply to report the first error it found in any Algol source program and then stop. (This was in line with his attitude to programming and programmers, but not very appropriate for the kind of industrial program development environment that we had in mind.) He in fact had a fear of producing a compiler that simply proceeded regardless with the compilation, or worse attempted to diagnose and correct things, after detecting a source program error, and then got confused and "discovered" all sorts of spurious errors. However, we came up with a scheme that almost entirely avoided producing incorrect error reports while going on and looking for further actual errors in the source program.

The basis of our error checking scheme was the assumption that the statement bracket structure of program was correct. This assumption cannot be verified until the end of the program text is reached, when it should be possible to confirm there have been an equal number of "begin" and "end" symbols. If the final error message produced in fact states that the statement bracket structure is incorrect, any previous error messages should be regarded with suspicion. But if the bracket structure is correct, it is possible to produce a list of errors which, though not exhaustive, is unlikely to contain any spurious errors (i.e., apparent inconsistencies arising from earlier errors). This is done by treating each block separately. When an error is found in a block, the rest of the block is skipped.

Any internal blocks are checked, but it is assumed that valid declarations for any nonlocal variables exist and might have been missed during the skipping of the block which contained an error. Since the outer blocks are not affected by the errors in an inner block, it is possible to resume normal checking after the end of a block containing an error has been reached. This scheme should find the first error in each block, a considerable improvement on finding just the first error in the program.

We documented this error-checking scheme in a report, which we sent to Dijkstra.[2] His reply, for the first time ever, started "Dear Brian"—I felt as if I'd just been awarded a higher degree. In what follows I will, wherever appropriate, refer to him similarly.

[2] Sections 1.4.2.1 and 3.4.7 of our book [Randell and Russell 1964] provide a full description of this scheme.

When we started implementing our compiler, there was not very much published technical literature to go on, and most of what was available concentrated on syntax analysis and especially the problem of compiling arithmetic expressions into conventional machine code. We had easily decided to base our translator on the technique that Dijkstra and Zonneveld had developed and used that involved assigning priority levels to all the language delimiters, not just the arithmetic operators [Dijkstra 1961]. These priority levels were then used to control the temporary reordering of the delimiters, with their priority levels attached, into a "translator stack." Such a translator stack was a holding store that enabled the Algol program text to be reordered into the "Reverse Polish" form required by the object program and to deal with Algol's nested statement structure. (In summary, when a delimiter was reached it was placed in the stack, after first unstacking and adding onto the partially generated object program appropriate code corresponding to any delimiters whose priority levels were not less than that of the current delimiter. Dijkstra used the analogy of a railway "shunting yard" to explain this technique. (He portrayed this shunting yard in the form of a "T" junction, with the translator stack as the branch line to perform the "shunting" or reordering.) [Dijkstra 1961].)

An alternative to this "bottom–up" parsing technique that we would have liked to try and perhaps use was the "top–down recursive descent" technique. However, no suitable implementation language, in particular a language supporting recursion, was available to us. Moreover, though we knew of the technique, we had no experience of actually using "bootstrapping" to produce a self-compiling compiler. In fact, we programmed the entire translator and the runtime control routines entirely using the KDF9 assembly language. This was all that was available for KDF9 at the time. And we employed a variant of the sort of logical flow diagrams that were commonly used for documenting DEUCE (machine-code) programs to document our detailed designs.

The other important data structure used by our translator was the "name list." This was used to contain, as the translator processed the Algol source program, the names and details of all the identifiers with a currently valid declaration. This was searched by the translator at each occurrence of an identifier in order to obtain and make use of details of its declaration in generating the appropriate object code. However, only the part of the name list appertaining to the current block or procedure was searched because at the use of an identifier it cannot be assumed that this identifier is nonlocal merely because its declaration has not yet been reached. Details of such identifier use occurrences were added to the name list for processing when and if a declaration was encountered. When the end of a block or procedure was reached, "declaration entries" could be discarded, and "use entries" moved, if necessary, into the section of the name list corresponding to the enclosing block. When the end of the program was reached, the only remaining "use entries" should be for standard library functions.

This strategy constructed the object program using a single scan of the source program but at times, when identifier uses preceded their declarations, would add incomplete ("skeleton") operations to the growing object code. These would be filled in later when the missing declaration information was reached. (Many years later, I found out that the X1 Algol compiler had in fact performed two scans of the source program and had filled in missing addresses by means of a load-time scan of the object code [Kruseman Aretz 2003].) Our single scan strategy proved to be very effective and so fast that Algol programs loaded from paper tape—in contrast even to KDF9 assembly language programs—were ready for use as soon as the tape had been read in (at full tape speed).

Clearly, the assignment of appropriate priority levels was crucial to the correct functioning of the translator. In July 1962, Lawford and I took time out to attend a week-long Programming Systems Symposium at the London School of Economics organized by Peter Wegner. At this symposium Peter gave a lecture on the "Burroughs Compiler Game," a board game produced by Burroughs for advertising purposes to demonstrate how their very novel new B5000 stack-oriented computer worked. In the game, cards bearing Algol symbols were moved around the board in accordance with a simple set of rules that implemented a priority level scheme showing how a conventional arithmetic expression was translated into Reverse Polish notation. Soon after Peter reached the point in his lecture when he started demonstrating a "game," I became aware that Lawford had spotted something. He casually suggested a further simple test expression to Peter, and then watched as Peter, and the game, got hopelessly muddled up. (He had noticed that the game's priority level rules did not cope properly with unary minus!)

I assume we continued to correspond with Edsger as we designed our compiler, though the interaction concerning compile-time error checking is the only one I now recall. The KDF9 computer was still being designed, and we made quite a lot of use of a DEUCE-based KDF9 emulator, though this emulator was many orders of magnitude slower than KDF9 was designed to be. Then we started gaining occasional overnight access to the first KDF9, at English Electric's plant at Kidsgrove, some 70 or so miles away. So, we oscillated back and forth between using DEUCE and KDF9, over a period of some months, not helped by the fact that the two machines represented binary numbers in differing directions, one with the most significant digits leading and the other with them trailing! During this period, we were delayed on a number of occasions as a result of discovering a discrepancy between the actual and the emulated machine. (When this happened, we would gleefully hand both "machines" back to their respective designers until the discrepancy was resolved.)

In retrospect, it is now clear that we greatly benefitted from the months of annoying delays to the completion of the first KDF9 computer. This was a period during which we continually refined our large set of very detailed logical flow diagrams documenting the design. Moreover, several colleagues started, both in and beyond Whetstone, using our design to develop compilers for a number of different computers, and providing us with lots of extremely useful feedback.

In fact, before we had got started on debugging our compiler, someone had startled us with the suggestion that we write a book about the compiler's detailed design. We had of course asked Edsger what he thought of this. He was very supportive, and gave us some very valuable, albeit uncomfortable, advice. This was that instead of just describing our compiler, we should try to list all the alternatives we had considered for each design choice and explain the basis on which we had chosen amongst these alternatives. Moreover, he advised us that we should make a point of admitting it when our choice had been arbitrary and/or had in retrospect proved to be wrong. I have always been extremely grateful for this advice—and have taken care to pass it on to all my graduate students. (One consequence of the advice was that we included in our book a fairly detailed comparative analysis of the existing literature on programming language—especially Algol—translation techniques, such as it was at that time, and in particular on techniques for translating arithmetic expressions, the one topic on which there was a relatively extensive literature.)

The several drafts of our book also received a great deal of scrutiny from the colleagues who were producing the other compilers. One result of their feedback that I recall concerns the text describing our version of Dijkstra's "Stack" and "Display" mechanisms.[3] The Stack is a last-in, first-out means of storing the sets of data (scalar variables and dynamic arrays) defined by declarations in blocks and procedures. Multiple copies of such sets of data might exist as a result of recursive procedure calls. However, at any one time, because of Algol 60's "scope rules," many of these sets of data might be *inaccessible*. (Identifiers defined in a block are local to that block and have no existence outside it.)

A linked list called the "Static Chain" is used to chain together the sets of currently accessible data in the Stack; a second linked list (the "Dynamic Chain") links all the sets of currently existing data, whether or not this data is currently accessible. The "Display" is an index that at all times mirrors the static chain, that is,

[3] Section 2.2.1.1 of our book [Randell and Russell 1964].

holds the storage locations in the Stack of the sets of currently accessible data, and so provides a direct means of addressing this data. The Display has to be carefully updated by a rather intricate algorithm that efficiently ensures that it remains valid as blocks are entered and left, procedure calls and returns occur, complicated parameters are accessed and evaluated, and so on. All data accesses, for example, for scalar variables involved in arithmetic expressions, can then be made directly, that is, without any linked-list searching. The Dynamic Chain is used for block, and in particular procedure, entry and exit.

This Stack and Display architecture was perhaps the most crucial and novel, and certainly the most arcane, part of the Whetstone runtime system—its description was completely rewritten some four or five times in response to our colleagues' suggestions and complaints. (I was amused to discover afterwards that the text in this potentially most difficult section of the book had ended up with a considerably lower "Fog Index" measure of comprehensibility [Gunning 1952, p. 289] than that of all the other sections of the book.)

Edsger's advice to us was of course in line with the fact that he always set himself and followed very high standards for clarity and presentation of writing and lecturing. Although the lecturing style that he developed had its critics (some of whom mistakenly interpreted his reflective pauses as mere theatricality), my own regret is that I cannot match either the clarity with which he lectured or the skill he demonstrated throughout his career at inventing or choosing appropriate problems and examples.

But he gave us some further advice that we did not in fact follow. This was to try and persuade our publishers that our book should be printed directly from typescript. (He had experienced considerable difficulty getting some of his manuscripts typeset to his satisfaction, not least because of confusions between the digit "1", and letter "l", and between the capital letter "O" and the digit "0".) I'm rather proud of our alternative, and in fact very effective, strategy. This was to have several typewriters, which had been made by the Leicester-based Imperial Typewriter Company, converted by them so as to have italicized numerical digits. Subsequently, the publisher's typesetting process proceeded smoothly and accurately.

When we sent the completed manuscript to Edsger, he kindly provided the following (excessively modest) text for its Foreword:

> I am very happy that the authors of this book have been so kind as to ask me to write a foreword to it, because the presence of their manuscript on my desk relieves my somewhat guilty conscience in more than one way. Accidental circumstances made me play some role in the initial stages of that part

> of their work to which they refer as 'the Whetstone Compiler', a fact which I would regret forever if their acquaintance with our solutions would have withheld them from looking for better ones themselves. The manuscript of this book, however, has convinced me that there is no reason for such regret. On the contrary, both the structure of the object program and the structure of the translator are of such an illuminating perspicuity that one can only be grateful when one's earlier work has been allowed to act as a source of inspiration.
>
> The second reason for my guilty conscience is related to the book itself. When they were urged to write it, they were so imprudent as to ask my advice, whether I thought that they could write such a book [. . .] I answered in a most encouraging and confirmative way, all the time aware of the fact that the preparation of a manuscript like this would be a tremendous task. The production of a piece of mathematical writing is always a considerable task which puts great demands on one's accuracy and ability to be concise but not obscure. Algorithmic translators, however, are hardly a standard subject of scientific publications, and the authors, who had to create a considerable portion of their 'metalanguage' to describe their subject, must have suffered greatly from this lack of a tradition [. . .] most of the difficulties met during translation can be traced down to a single source, viz. either an awkward feature of the language or an awkward feature of the machine [. . .] they make translators unnecessarily expensive to construct, unnecessarily expensive to run and, worst of all, they tend to affect the reliability and trustworthiness of the whole system. If we agree on this, the art of language designing becomes the art of designing a powerful and systematic language primarily from those elements which can be processed elegantly by a translator—and this book shows that this set is certainly not empty—and the art of machine designing becomes the art of building-in such features that can be used by a translator in a neat and orderly way—and the Control Routine described in this book shows some of these features. Many of the signposts along the road to improvement in the computer field are set by the experiences gained by the implementors.

Edsger of course had a continuing interest in the interplay between language and machine design. He was particularly interested in the Burroughs B6500 architecture, as the following translated extract from his trip report EWD286 [Dijkstra 1970] makes clear:

> From London I travelled with Brian Randell [. . .] Thursday afternoon and Friday we visited the Burroughs factory, Pasadena together [. . .] My overall

> impression of this group is that there are some excellent people, people who know what they are doing and why they are doing it, who have a lot of fun with each other, but don't always have it easy with their management.
>
> [Burroughs started their B6500 project] with a consistent design and that didn't suffer too much during the development phase. (One of the less appealing things about this machine, in my opinion, is the way physical addresses can diffuse through the stack; when I saw this, my first reaction was "What a pity I wasn't involved in this design early on. This might have been prevented."
>
> [. . .] We then spent six nights on Catalina Island for a five-day gathering in which twenty of Burroughs' top designers were exposed to outside influences. And I was such an outside influence. The working hours were from nine to twelve and from three to six and a few times in the evening. It took place in the patio of a Spanish-style "Country Club", people sat in the sun or under umbrellas according to taste and skin.
>
> I spoke to them for about six hours and with a few exceptions (hardware designers from Paoli) a lovely audience, which gave me a clearer picture of the relevance of my story. I have usually listened with great pleasure to the other "outside influences", they were very varied. For example, there was a Fairchild man on integrated circuits; this was a very streamlined story, complete with slides and all. A bit of a "slick commercial" but if you saw through that, it gave an impression of what awaits us in the hardware field. [There were] three very different stories by Brian Randell, one about memory management, which I know well and don't really like, one about reliability, and finally—all to broaden the view—a story about Ludgate, an Irishman, who at the beginning of this century between Babbage and Aitken, wanted to make an automatic computer; this Irishman was completely forgotten and Brian only discovered him a month or so ago.

I have very fond memories of this Catalina Island gathering, but have only recently discovered that Edsger had documented it in his EWD286 trip report. The "lecture room" was I recall a veritable tropical paradise—for example, one of his talks was rudely interrupted by a beautiful hummingbird, something neither of us had ever seen before. The whole event, which had been put together by Bob Barton (of B5000 fame),[4] was highly enjoyable. It was not just intellectually stimulating but also notable for a great amount of good-natured teasing and practical joking throughout.

[4] Robert S. Barton: https://en.wikipedia.org/wiki/Robert_S._Barton.

I now realize it must have been on this trip that, at his suggestion, Edsger and I began to work together on trying to clarify and document how the B6500 stack architecture could be improved. I remember that we produced just two or three draft pages of our intended document. Unfortunately, these pages seem not to have survived, and I cannot now recall the details of our planned improved stack architecture, though I now presume we were trying to deal with his concerns about the use of physical addresses in the stack. Incidentally, on a further visit I paid to Burroughs in Pasadena, I learned that it was Bill McKeeman, then of UC Santa Cruz, who had been their main consultant on the design of the B6500 stack architecture. (His more general recommendations concerning "language-directed machine architecture" are documented in McKeeman [1967].) Apparently, it was only later that the Burroughs designers found that the detailed design proposals he had made to them were based closely on the Whetstone Algol Object Program and were fully described in Randell and Russell [1964].

It is however interesting, and to my mind somewhat regrettable, that stack mechanisms and language-directed machine architectures have not in fact caught on. Instead, as explained in Hennessy and Patterson [2019], the trend has been in the opposite direction, toward lower-level instructions that are fast and well suited for efficient code generation and optimization (RISC).

26.5 1964—IBM and Algol WG 2.1

By 1964, our Whetstone KDF9 compiler, and a number of other versions of it, had been completed, and our book had been published [Randell and Russell 1964]. To my surprise, I was invited to join the IBM Research Center, in Yorktown Heights, NY. (I had not thought of what we were doing as research—we produced our compiler because it was urgently needed.) With some misgivings, which I'm sure Edsger shared, since to him IBM was "The Great Satan," such was his dislike of their hardware and software, I joined IBM Research—and was able at last to take up my invitation to membership of the IFIP Working Group 2.1 on Algol. (I hadn't been able to get permission from English Electric to accept—one trip abroad was enough in their view.). By this time Edsger was also a member—and from then on, for many years at the Algol committee and elsewhere, I had numerous opportunities to meet him—and to have numerous memories to draw on for this account.

The first WG2.1 Algol Committee meeting that we both attended was held in Baden, Austria, in 1964—I had just left English Electric and attended as an IBM research staff member, though I had yet to reach the T.J. Watson Research Laboratory in Yorktown Heights and actually "sign-in." The meeting was organized in two parts, either side of the IFIP Working Conference on Formal Language Description

Languages, a conference that was very ably organized by Heinz Zemanek and his IBM Vienna Laboratories team. This Conference was an attempt to bring together people working on formal languages with people working on programming language definition and compilation. In fact, there was little meeting of the minds, and I must confess my main memories of the Conference center on the brilliantly funny and sarcastic after-dinner speech, "Our ultimate meta-language" [Duncan 1966]. This was given by Fraser Duncan, a colleague from English Electric Kidsgrove, who was overall manager of the company's KDF9 compiler developments.

My vague recollection is that neither Edsger nor I took a particularly active part in the Conference. In fact, my main contribution was to get Dr. Zemanek's assistant, Norbert Teufelhart, who was very ably supervising all the conference arrangements, to pour enough whisky into Fraser Duncan to persuade him to go ahead after all with his invited speech. (Fraser had become so disenchanted with the tenor of the discussions at the conference that he had become unwilling to give his speech. It became evident that a considerable amount of alcohol had been involved in successfully overcoming Fraser's reluctance.)

Subsequently, Tom Steel, the Editor, insisted on including a full transcript of a recording of the resulting speech in the Conference Proceedings despite Fraser's embarrassed attempts to bowdlerize it. This is fortunate since in his speech Fraser had used clever humor to convey a number of uncomfortable technical truths, indeed truths of continuing validity. For example, he had pointed out that there had been much disagreement on the meaning of the phrase "Algol-like," and the only thing that everyone had agreed on was that Algol was not an Algol-like language!

Before I joined IBM Research, I had made it clear I had no intention of writing another compiler. Instead, I got involved in research in computer architecture and in particular in operating systems—as had Edsger. Indeed, before he got involved in compilers, he had worked on communications facilities and (the entirely novel, and to his view of programming, fearsome idea of) interrupt handling. This was for the Electrologica X1 computer, work for which he obtained a Ph.D.—work that provided a basis on which he laid many of the foundations of the entire subject of concurrent programming.

26.6 1967—SOSP

Dijkstra's seminal contributions in the area of concurrent programming were on problems such as process synchronization, critical sections, and system deadlocks—for example, his famous work on the "dining philosophers" problem. He, and this work, starred at the ACM's first Symposium on Operating System

Principles (SOSP), where he presented a paper on the THE Operating System—see below. I also attended this conference, which was held in Gatlinburg, TN, at the time a so-called "dry" town, in which the only alcohol that could be obtained was very appropriately called "near beer"—whose alcohol content was as low as its temperature, and as Edsger's spirits when he found that it was all that was available.

At the Symposium my recollection is that Edsger used a very simple and elegant diagram (representing what he termed "progress space") showing the dangers of deadlock between a pair of processes competing for unsharable resources—the term he used at this time for deadlock was "deadly embrace," and his impact was such that the symposium organizers made an impromptu award to him of two stuffed toy "Smoky Mountain Brown Bears" locked in a deadly embrace. But he also had one other major impact on at least some of the conference attendees—and this was quite unintended.

There had been a move to set up a committee to define an ideal list of operating system primitives. Edsger and I had both refused invitations to join this committee, with some alacrity. That evening he and I had dinner together, during which we had great fun drawing up a spoof list of desirable operating system facilities—a list that I know Edsger carried around in his wallet for some years afterwards. (The only item I can remember from the list is "Execute Operator.") Edsger was so pleased with our list that he took it across to a group of conference attendees at a table on the far side of the dining room, only to return looking somewhat abashed—it turned out that this group was the committee, having its first meeting!

After Gatlinburg, Edsger travelled back with me to IBM Research, where I had arranged that he would give a lecture in the large and very impressive research auditorium. After his talk I confessed to him that I had planned a practical joke, but that at the last moment I had had cold feet and not gone through with it. This was to place across the top of the lectern, invisible to the audience and to Edsger until he came up to the lectern following my introduction of him, a plastic strip I had taken from my hotel room in Gatlinburg. The strip said: "This has been sanitized for your protection." Such a warning would have been very appropriate, given his attitude to IBM, but when I afterwards confessed what I had been planning, he thanked me for my restraint, saying that it would have taken him the whole time allotted to his lecture to stop laughing.

26.7 The THE Operating System

Edsger's paper on the THE Operating System at the Gatlinburg symposium was published in the *Commun. ACM* in 1968 [Dijkstra 1968]. (THE stands for "Technische

Hogeschool [Technological University], Eindhoven," where in 1962 he had taken up a professorship.) The THE system was a staggering achievement because it was so well designed and coded that there were virtually no bugs in it. One of the main keys to this was its structure as a set of levels of abstraction, each of which hid some aspects of the underlying hardware and introduced further convenient facilities for use by user programs. The operating system had a five-level structure: level 0 dealt with processes and interrupts, level 1 with storage allocation, level 2 with communication with the operator's console, level 3 with I/O devices, and level 4 was where user and support programs (e.g., the compiler) resided. The choice and ordering of these levels were not arbitrary—rather what turned out to be critical was that at each successive level the time granularity (i.e., the average length of the intervals between events) became roughly an order of magnitude larger. As a result, one could to a great extent design these levels independently of each other.

All this was highly revolutionary. From then on for years it was almost obligatory, in any new paper on operating system design, to reference this paper. Now I fear there are generations of computer scientists who have never heard of it. I urge all members of these generations to read it.

I am very proud of the fact that at IBM Research I and a colleague, Frank Zurcher, were by coincidence already working along rather similar lines at this time, and even using the same term "levels of abstraction." In our case we were motivated by the possibility of using simulation to help evaluate and guide the progress of large hardware/software system designs, and we investigated means of constructing systems made of actual coexisting separate levels of abstraction [Zurcher and Randell 1968]. (Dijkstra's use of levels of abstraction was in contrast just descriptive.) This work in fact had, and to my surprise continues to have, a significant effect on my subsequent researches in the area of system dependability.

Incidentally, towards the end of my time at IBM Research, I stumbled across an internal "Competitor Analysis Review" report, I think it was called. This was a lengthy restricted document that provided a detailed comparison of the English Electric KDF9 computer against the most comparable IBM computer—I forget which one. The report had separate chapters on various different architecture and technology topics, almost all of which ended in comparisons that were noticeably favorable to the KDF9. However, there was a final chapter comparing the two companies' resources and general competency. The report's overall conclusions were simply that the information in this last chapter was such as to indicate that, despite all the comments in the earlier chapters, the KDF9 did not constitute a credible threat to IBM. Needless to say, I wish I had been able to show this report to Edsger.

26.8 The 1968 NATO S/W Engineering Conference

In October 1968 Edsger and I participated in the first of the NATO Software Engineering conferences, held at Garmisch-Partenkirchen, in Germany [Naur and Randell 1969]. We have both since gone on record as to how the discussions at this conference on the "software crisis" and on the potential for software-induced catastrophes strongly influenced our thinking and our subsequent research activities. In Edsger's case it led him into an immensely fruitful long-term study of the problems of producing high quality programs. In my case, it led me to explore the then very novel, and still somewhat controversial, idea of "design fault tolerance." Suffice it to say that our respective choices of research problem suitably reflect our respective skills at program design and verification.

26.9 Algol 68

The year 1968 was also a critical one in the life of the IFIP Algol Committee, which was striving to develop a successor to Algol 60. By this time, Niklaus Wirth had left the committee to pursue his own ideas on language development, work which later resulted in the Pascal language. Dahl and Nygaard had produced Simula (an extension of Algol 60 aimed at simulation) and then Simula 67, their "Common Base Language" [Dahl and Nygaard 1967], the language that was the foundation of what became known as object-oriented programming. The Algol Committee had become dominated by Aad van Wijngaarden and was pursuing two issues in parallel—language development and techniques of language description—with van Wijngaarden pushing his ideas on "two-level grammars" [van Wijngaarden 1965].

A series of ever more fractious meetings was held, culminating in one in Munich in late 1968, when a majority of the committee voted to approve the Algol 68 Report [van Wijngaarden et al. 1969]. However, Dijkstra was one of the leaders of a gang of eight (of which I am proud to have been a member) that produced a Minority Report, arguing that the Algol 68 Report should not be approved [Dijkstra et al. 1970]. In fact, I recall we felt that Simula 67 was a more fruitful development—to my subsequent regret we did not actually say this in our Minority Report. The committee then split, and a group of us left the Committee, and later founded a new IFIP Working Group on Programming Methodology, though we regarded this group, WG 2.3, as having twin roots, the Algol Committee and the NATO Software Engineering Conference.

WG 2.3 became an important forum, one at which Dijkstra tried out, polished, and promoted many of his highly influential ideas on methods of developing high quality programs. (I remained with this working group for a number of years, though later left as my interests turned increasingly elsewhere.)

26.10 The 1969 NATO Conference

There was a further, much less successful, NATO Conference on Software Engineering in 1969 in Rome, which Edsger and I both attended [Buxton and Randell 1970]. I've mentioned that Edsger could give very clear and elegant lectures. However, he was rarely willing to make any concessions to his audience, for example in adopting any of the terminology that they might already be familiar with, and so his lectures were not always well-appreciated. This was the case in Rome, and one attendee in particular, Tom Simpson of IBM Houston, justifiably famous within IBM for his pioneering work on operating systems, in particular the HASP (Houston Automatic Spooling Priority) System, started an argument with Edsger. This argument continued at intervals throughout the conference—and I watched with fascination as Tom and Edsger gradually learnt how to communicate with, and to gain great respect for, each other. It is a pity that some of the other people who then, and in the years to come, reacted negatively to Edsger, did not have the time or inclination to get to appreciate him properly. I'll return to this point below.

26.11 In 12 Months from June 1974

The year 1968 was a very eventful year in my and Edsger's life. However, at the BCS meeting in November 2002 in memory of Edsger, Tony Hoare chose another 12-month period in Edsger's life to illustrate how amazingly creative he was. This was the period from June 1974, which Tony illustrated by referring to a sample of the memos produced in this period. Edsger produced well over a thousand EWD memos, as Edsger called them, that were distributed Russian "samizdat" style for years by a worldwide network of colleagues. So prolific was he that at least one person, not noticing that they were Edsger's initials, wrote to him asking if "EWD" stood for "Eindhoven Working Document." Most of them are now available from the E.W. Dijkstra Archive at http://www.cs.utexas.edu/users/EWD/.

The EWDs that Tony highlighted were as follows:

EWD418—on "Guarded commands, nondeterminacy and formal derivation of programs"—this led to his book *A Discipline of Programming* [Dijkstra 1974a].

EWD426—"Self-stabilizing systems in spite of distributed control"—which created a whole subculture of computing science, addressing the issue of how certain kinds of systems might be designed so as to be inherently capable of recovering after errors have occurred and propagated far and wide [Dijkstra 1974c].

EWD464—"A new elephant built from mosquitos humming in harmony"—this launched the whole new class of highly parallel algorithms now known as "systolic algorithms," since data pulses through them rhythmically, rather like blood flows through the heart [Dijkstra 1974b].

EWD492—"On-the-fly garbage collection"—again a paper that prompted a whole series of follow-ups by others, addressing the incredibly tricky problem of identifying and gradually catching up with the production of garbage (data items that had become unlinked and so no longer accessible) without stopping the system in order to tidy it up [Dijkstra 1975a].

This truly is an astonishing output, in just one year—yet these are just four of the over 70 EWDs that Edsger wrote and circulated during this period.

26.12 Some EWD Interludes

The EWD series includes a considerable number of Edsger's trip reports, including ones to the International Seminars on the Teaching of Computing Science at University Level that were held annually at Newcastle University from 1968 to 2001, sponsored for many years by IBM and later by Amdahl and then ICL. These four-day seminars brought together an invited audience of senior UK and European computing academics to hear a series of presentations from distinguished international speakers (of our choice). Each seminar was on a different theme, usually a major computer science research area. Edsger attended a number of these seminars and commented acerbically afterwards on them in his subsequent trip reports, leaving no doubt as to which lectures he strongly approved of and which he greatly disliked. He gave lectures himself at four of our seminars, including the 25th Anniversary Seminar. (All eight of the speakers at this celebratory seminar had lectured at one or more earlier seminars, and seven were, or would shortly be, Turing Award winners—Dijkstra, Hoare, Knuth, Lampson, McCarthy, Nygaard, and Rabin.)

Edsger stayed at Hotel Randell each time, so became well-known to our growing family. Our daughter in particular evidently benefited from his visits. I recall that on one early occasion, when she was in the back of my car, I stopped at an intersection and signaled to an oncoming driver who wanted to turn across in front of me that he should go first. (I'd also wanted to turn across in front of him.) A small voice asked: "Why are you waving to the other driver, Daddy?" I explained why. After a short pause our daughter asked: "What if you'd both waved at the same time?"

I said we might both start forward then we'd realize, and stop, then one would wave at the other to go first. Silence—then: "I don't think that works." Needless to say, I greatly enjoyed reporting this incident to Edsger.

She remembers Edsger's visits to Newcastle vividly: "I recall his super accurate and precise English—which was at least as good as that of my parents—and how I and my young brothers watched Edsger's methodical deliberations at breakfast with fascination, as we nudged each other into silent bated breath." But her strongest memory was of the occasion when: "Dad was showing little interest in a particularly stressful maths homework crisis. Mum encouraged me to ask Edsger for help. I was so desperate I finally plucked up courage to ask. I remember sitting together with him poring over the problem in a quiet, darkened room, and being on tenterhooks because of the deep breathing and long silences as Edsger considered the maths problem. I assume he did succeed in helping me, but actually the strong memory I retain is of his complete, absorbed attention and concentration, as well as the terrifying, heart-stopping pauses."

I very much sympathize with this comment. During his visits to our home, Edsger liked to relax by playing on our (baby) grand piano and our harpsichord. This I'd built from a Zuckerman harpsichord kit, which we'd brought back with us from America. I'd then spent two happy Newcastle winters assembling it. However, I was still nervous about what expert harpsichordists might think of the results of my labors. Unfortunately, Edsger had a habit of every now and again suddenly pausing in mid-performance, just as in his lectures. These pauses used to worry me greatly until he and I reached a formal agreement that if ever he stopped because he felt there was a problem with my harpsichord, he would tell me immediately. Thereafter I could relax and calmly ignore his contemplative pauses.

My own most direct involvement with the EWDs arose out of the little series of satirical EWD documents that Edsger wrote in the persona of "Chairman of Mathematics Inc.," someone who evidently was keen to monetize all scientific discoveries and who looked at science as an economic commodity. Edsger used these documents as a means of implicitly ridiculing the programming practices of certain large industrial software projects. Indeed, these Mathematics Inc. EWDs were of course very much part of his career-long campaign for programming to be recognized and practiced as a branch of mathematics, one that prioritized elegance and beauty rather than as an engineering-style massed battle with complexity and enormity.

I was one of the recipients of EWD475: "A letter to my old friend Jonathan" [Dijkstra 1975b], a Mathematics Inc. document that he later included in his book *Selected Writings on Computing: A Personal Perspective* [Dijkstra 1982]. This was a request to the Chairman's former classmate, who had been a law student, for help with the legal protection and commercial exploitation of Mathematics Inc.'s latest product, a proof (now almost complete) of Riemann's Hypothesis, which the Company had recently rebranded as Riemann's Theorem. I decided that it was appropriate for Jonathan to reply positively and took advantage of a document I had in my files from the National Research Development Corporation (NRDC). I had recently been involved in negotiations with them over a patent that I and colleagues had been awarded on an invention resulting from a UK Government-funded research project, so our patent had to be assigned to the NRDC. The document I had was their standard Revenue Sharing Agreement.

I had this lengthy and very legalistic document carefully retyped, with the word "inventor" replaced throughout by "mathematician," and "invention" by "theorem." I obtained some appropriate letterhead for Jonathan, who I knew now worked for the legal firm of Obfuscate, Lobachevsky, Fudge and Partners, of 13 Shyster Lane, Oldcastle upon Time. I then used this for a cover note explaining the terms under which it would be possible for Jonathan to assist Mathematics Inc. The following evening, I received a telephone call at home from Edsger, who was literally helpless with laughter. Later I received a copy of EWD539 [Dijkstra 1975c], "Mathematics Inc., a private letter from its Chairman", which described in glorious detail the actions that his Company had taken as a result of receiving the letter from Obfuscate, Lobachevsky, Fudge and Partners, and attached a copy of the Revenue Sharing Agreement, which he described as having been very helpful—it had given them "all the information we wanted to have!" (Much later the Dijkstra Archive was provided with, and made available, the originals of both the cover letter and the Agreement [Pettifogger 1975].)

26.13 The 60th Birthday Salute

In 1989, as Edsger's 60th birthday approached, I and a number of his friends and colleagues were approached and asked to provide contributions to a book that would be produced as a surprise in his honor [Feijen et al. 1990]. I was concerned that my field of research, on fault tolerance and, in particular, my attitude that design faults often had to be tolerated, were so far from his interests and attitudes that a contribution from me would not be appropriate or appreciated. I was very kindly assured by the Editors that this was not at all the case and was instead invited and agreed to write a Foreword to the book. This was a naive decision on my part since this proved much harder to write. However, I'd like to end this account, first by repeating my acknowledgment of the great debt I owe to Edsger, and then by using some of the remarks I made in this Foreword:

> Edsger is, with all that the word implies, a perfectionist, who expects as much of his listeners and readers as he demands of himself. His programming and his mathematics are strongly guided by his concern for clarity of notation and exposition, and indeed for what he quite justifiably terms "beauty." Thus, his descriptions of problems and solutions, both in his lectures and published papers, and in his EWD series of documents [. . .] are often vivid and compelling.
>
> Although over the years much of his work has had a very immediate impact, on debate if not always in practice, some of his more recent diagnoses and prescriptions have proved harder to take. This is in part because he is sometimes more concerned with the truth of his arguments than with whether they are couched in terms that will help to ensure that they have the desired effect on his audience. Nevertheless, careful study of all his writings is highly recommended to all who care for the future health of computing science.
>
> Much of the last phase of his career, at the University of Texas, where he went in 1984, was spent investigating the effective structure of logical arguments, applied to mathematics as much as programming—a distinction whose validity he denied, since to him programming was mathematics. This work set standards that many cannot even recognise, let alone aspire to. However, I am confident that it will eventually have deep and long-lasting effects, much of it indirect, through the inspiration that it provided to close colleagues.
>
> Let me end with a quotation from George Bernard Shaw [Shaw 1903]:
>
> > The reasonable man adapts himself to the world: the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.
>
> Edsger W. Dijkstra is, in many ways, just such a man. Needless to say, the world of computing science could well do with many more "unreasonable" men (and women) of his caliber!

References

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and P. Naur (Eds.). 1960. Report on the algorithmic language ALGOL 60. *Commun. ACM* 3, 5, 299–314. DOI: https://doi.org/10.1145/367236.367262.

J. N. Buxton and B. Randell (Eds.). 1970. *Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee*, Rome, Italy, 27–31 October 1969. Scientific Affairs Division, NATO, Brussels. Reprinted in *Software Engineering: Concepts and Techniques* (J. M. Buxton, P. Naur, and B. Randell [Eds.]). Petrocelli/Charter, New York, 1976. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

O.-J. Dahl and K. Nygaard. 1967. *SIMULA 67: Common Base Definition*. Norwegian Computing Center, Oslo, Norway.

G. Davis. 1960. The English Electric KDF9 Computer System. *Comp. Bull.* 4, 3, 119–120.

E. W. Dijkstra. 1961. Making a translator for Algol 60. *A.P.I.C. Bull.* 7, 3–11.

E. W. Dijkstra. 1968. The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5, 341–346. DOI: https://doi.org/10.1145/363095.363143.

E. W. Dijkstra. April. 1970. Verslag van mijn reis naar California. EWD286. In Dutch. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD286.PDF.

E. W. Dijkstra. June. 1974a. Guarded commands, non-determinacy and a calculus for the derivation of programs. EWD418. Published as: Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8, 1975, 453–457. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD418.PDF. DOI: https://doi.org/10.1145/360933.360975.

E. W. Dijkstra. November. 1974b. A new elephant built from mosquitos humming in harmony. EWD464. Published in: *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982, 79–83. Available from E. W. Dijkstra Archive, https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD464.PDF. DOI: https://doi.org/10.1007/978-1-4612-5695-3_14.

E. W. Dijkstra. June. 1974c. Self-stabilizing systems in spite of distributed control. EWD426. Published in *Commun. ACM* 17, 11, 643–644. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD426.PDF. DOI: https://doi.org/10.1145/361179.361202.

E. W. Dijkstra. April. 1975a. On-the-fly garbage collection: An exercise in multiprocessing. EWD492. Published as: E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11, 966–975. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD492.PDF. DOI: https://doi.org/10.1145/359642.359655.

E. W. Dijkstra. February. 1975b. A letter to my old friend Jonathan. EWD475. Published in: *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982, 99–103. E. W. Dijkstra Archive, https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD475.PDF. https://doi.org/10.1007/978-1-4612-5695-3_18.

E. W. Dijkstra. December. 1975c. Mathematics Inc., a private letter from its Chairman. EWD539. Published in: *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982, 184–187. E. W. Dijkstra Archive, https://www.cs.utexas.edu/users/EWD/ewd05xx/EWD539.PDF. DOI: https://doi.org/10.1007/978-1-4612-5695-3_32.

E. W. Dijkstra. 1982. *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag. DOI: https://doi.org/10.1007/978-1-4612-5695-3.

E. W. Dijkstra, F. G. Duncan, J. V. Garwick, C. A. R. Hoare, B. Randell, G. Seegmüller, W. M. Turski, and M. Woodger. March. 1970. News item—Minority report. *ALGOL Bulletin,* *AB31.1.1*. http://archive.computerhistory.org/resources/text/algol/algol_bulletin/A31/P111.HTM.

F. G. Duncan. 1966. Our ultimate metalanguage: An after-dinner talk. In T. B. Steel (Ed.), *Formal Language Description Languages for Computer Programming*. North Holland, 295.

W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra (Eds.). 1990. *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*. Texts and Monographs in Computer Science. Springer-Verlag, New York. DOI: https://doi.org/10.1007/978-1-4612-4476-9.

R. Gunning. 1952. *The Technique of Clear Writing*. McGraw-Hill.

J. L. Hennessy and D. A. Patterson. 2019. *Computer Architecture: A Quantitative Approach* (6th edn.). Morgan Kaufmann.

B. B. Kristensen, O. L. Madsen, and B. B. Jensen. 1974. A PASCAL environment machine (P-code). *DAIMI Report Series* 3, 28. DOI: https://doi.org/10.7146/dpb.v3i28.6447.

F. E. J. Kruseman Aretz. June. 2003. *The Dijkstra–Zonneveld ALGOL 60 Compiler for the Electrologica X1*. Technical Report SEN-N0301. Centrum voor Wiskunde en Informatica.

W. M. McKeeman. 1967. Language directed computer design. In *Proceedings of the Fall Joint Computer Conference*. ACM, 413–417. DOI: https://doi.org/10.1145/1465611.1465665.

P. Naur and B. Randell (Eds.). 1969. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch-Partenkirchen, Germany, 7–11 October 1968. Scientific Affairs Division, NATO, Brussels. Reprinted in *Software Engineering: Concepts and Techniques* (J. M. Buxton, P. Naur, and B. Randell [Eds.]), Petrocelli/Charter, New York, 1976. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

J. Pettifogger. 1975. Letter to E.W. Dijkstra, Chairman: Mathematics Inc., 29. https://www.cs.utexas.edu/users/EWD/ewd04xx/Randell.pdf.

B. Randell. 2003. Edsger Dijkstra. In *9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS Fall 2003)*, 1–3 October 2003, Anacapri, Italy. IEEE, 1–10. DOI: https://doi.org/10.1109/WORDS.2003.1267483.

B. Randell and L. Russell. 1962. Discussions on ALGOL Translation at Mathematisch Centrum, W/AT 841, Atomic Power Division, English Electric Co., Whetstone, Leics. http://homepages.cs.ncl.ac.uk/brian.randell/Papers-Articles/34.pdf.

B. Randell and L. Russell. 1964. *Algol 60 Implementation*. Academic Press.

G. B. Shaw. 1903. *Man and Superman—“Maxims for Revolutionists: Reason”*. Constable.

A. M. Turing. 1945. *Proposals for the Development in the Mathematics Division of an Automatic Computing Engine (ACE)*. Technical Report E882. National Physical Laboratory. Reprinted with foreword by D. W. Davies, *NPL Report Comm. Sci*. 57, April 1972.

A. van Wijngaarden. 1965. *Orthogonal Design and Description of a Formal Language*. Technical Report MR 76. Mathematisch Centrum, Amsterdam.

A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. 1969. Report on the algorithmic language ALGOL 68. *Numer. Math.* 14, 2, 79–218.

F. W. Zurcher and B. Randell. 1968. Iterative multi-level modelling. A methodology for computer system design. In A. J. H. Morrel (Ed.), *Information Processing, Proceedings of IFIP Congress 1968*, Edinburgh, UK. North-Holland, 867–871.

27

Evoking Whitehead’s Dictum

Fred B. Schneider

> It is a profoundly erroneous truism, repeated by all the copybooks, and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of operations which we can perform without thinking about them. Operations of thought are like cavalry charges in a battle—they are strictly limited in number, they require fresh horses, and must only be made at decisive moments.
>
> [Whitehead 1958, ch. 5]

Edsger Dijkstra’s work on formal derivation of programs applies Whitehead’s dictum above to programming. Instead of mentally simulating program execution—what is called *operational reasoning*—Dijkstra showed how programmers could derive a program by manipulating assertions about program variables. A correctness proof would be produced as a side effect of these manipulations.

In his illustrations, however, Dijkstra did not hesitate to work outside of the formalism. His derivations were invariably compelling, so it was easy to overlook that some parts were not formalized. Where should we draw the line about what needs to be formalized? Most would agree that formalism makes ideas more precise but also can make those ideas less accessible to a broader audience. Therefore, the use of formalism makes sense when the additional effort required by the author and readers will yield benefits—because precision is important or because formal manipulations (manual or automated) would be helpful for drawing conclusions from a statement or for increasing our assurance in the validity of a statement. The formalisms he was using undoubtedly also influenced Dijkstra’s decisions about what to formalize.

This chapter ostensibly is about Whitehead's dictum, and what should get formalized. But there is a subtext—to describe one small instance of Dijkstra's enormous impacts on Computer Science, by telling how Dijkstra's research affected mine and how personal interactions with him affected me as a researcher.

27.1 Dijkstra's Influence: On Concurrent Programming

My first exposure to the idea that a program might be derived from its specification was *A Discipline of Programming* [Dijkstra 1976], which I read in spring of 1978. At the time, I was at Stony Brook completing a Ph.D. dissertation on operational reasoning for concurrent programs that synchronized by using monitors. Inspired by *Discipline*, I resolved to explore the use of weakest preconditions for the derivation of concurrent programs. I joined the Cornell CS faculty, not realizing that a member of the department, David Gries, was Dijkstra's close friend. I was also unaware that Gries and his Ph.D. student Susan Owicki had already developed an inference rule [Owicki and Gries 1976] for concurrent composition in Hoare's logic [Hoare 1969] of partial correctness triples $\{P\}S\{Q\}$. Dijkstra gives his account of the Owicki–Gries work in EWD554 [Dijkstra 1982].

The Owicki–Gries work enables *a posteriori* verification of an existing concurrent program. Its concurrent composition rule introduced proofs of *interference freedom* which, in retrospect, can be seen as proving that no process invalidates an inductive invariant[1] associated with another process. I conjectured that failure to satisfy an interference-freedom obligation could motivate steps in the derivation of a concurrent program, in analogy with how Dijkstra had used weakest preconditions along with the obligations associated with verifying a loop invariant when he derived assignment statements for the loop body and derived the Boolean guard that causes loop iteration to terminate.

The approach I envisioned used interference-freedom obligations and weakest preconditions for

- determining assignment statements that should be added to a program in order to expose control state,

[1] That invariant is: If execution starts at any control point with the associated assertion *true*, then whenever execution reaches another control point the assertion associated with that control point will also hold. This invariant is exactly what Floyd [1967] introduced with his method for verifying a program that is formulated as a flowchart. It is also the proof approach that Lamport [1977] independently and contemporaneously developed for *a posteriori* verification of concurrent programs described by flowcharts, where edges are labeled with assertions.

- determining synchronization statements that would be added to a program in order to delay execution that would violate interference-freedom, and
- identifying code segments whose execution in a program, if made mutually exclusive, would eliminate a problematic interference-freedom obligation.

I tried my approach and was successful in deriving concurrent programs for various standard examples from the operating system literature. Based on that experience—and with hopes of reaching a larger audience—I undertook to write a textbook: *On Concurrent Programming* (OCP) [Schneider 1997]. The book would describe a methodology for the derivation of concurrent programs, giving examples and general strategies. It would mimic what was done by Dijkstra in *Discipline* and later by Gries in his book *Science of Programming* [Gries 1981]. However, the formalization of interference-freedom needed *proof outlines*, which are Hoare triples where the assertion before and after each statement had been left in place. Proof outlines did support reasoning based on weakest preconditions, so proof outlines were the obvious choice for use in OCP.

The intended audience for OCP was people interested in writing operating systems. This audience tended to focus on mechanisms, as becomes clear from looking at any operating systems textbook. Moreover, computer scientists in those days believed that providing the right mechanisms would help programmers to avoid errors and allow compilers to detect certain programming errors. Dijkstra's [1968c] admonition against programming with **goto** statements was an early example of this argument. So, I decided that OCP should discuss the use of well-known synchronization primitives and their embodiments in programming languages, showing how each primitive could be used in discharging interference-freedom obligations. Dijkstra had articulated the computational model (processes that exhibit finite progress) that OCP adopted, and he had introduced many of the primitives (e.g., binary and general semaphores).

Dijkstra published relatively few derivations of concurrent programs. On-the-fly garbage collection [Dijkstra et al. 1978] was one of his important case studies, but this algorithm did not use synchronization primitives so it was less relevant for the intended focus of OCP. In contrast, his EWD703 "A tutorial on the split binary semaphore" [Dijkstra 1979] was highly relevant to the theme of OCP, as was its sequel [Dijkstra 1980] on translating a concurrent program that used general semaphores into one that used binary semaphores. These derivations, as well as the derivations of other concurrent programs in Dijkstra's writings, were based on maintaining an inductive invariant. Specifically, Dijkstra used weakest preconditions (i) to check whether a coarse-grained assignment statement could be replaced by a code fragment comprising finer-grained statements and (ii) to determine synchronization conditions for delaying execution in order to prevent falsifying the inductive invariant. Both of these strategies were among those that OCP would present.

To specify a concurrent programming problem, Dijkstra would posit a program skeleton, which gave assertions at some control points and had some code fragments that were unspecified. The program skeleton might include assignments to so-called *ghost variables*, variables that were needed for a problem specification or correctness arguments but not used to control execution of the program. For example, the value of a ghost variable *luck1* (a name used in EWD703 [Dijkstra 1979]) might be changed by assignment statements in the posited program skeleton, causing *luck1* (respectively, *luck2*) to be *true* whenever process 1 (respectively, process 2) is executing in its critical section. Then, for deriving protocols to ensure mutual exclusion of execution by process 1 and process 2 in their critical sections, it sufficed if every control point was associated with an assertion and each of those assertions implied $\neg(luck1 \wedge luck2)$. So the derivation involved adding or modifying code in order to strengthen assertions in the evolving program skeleton. The last step in the derivation was to delete ghost variables since (by construction) their values had no effect on execution.

Some of Dijkstra's steps were not formal. The reader of his derivations is never told what conditions must hold for the various rules he uses to be applied, nor is the reader told what kinds of specifications can and cannot be handled by the methods he is describing. For comparison, Dijkstra's work on the derivation of sequential programs expressly targeted total correctness specifications (i.e., execution started in a state satisfying a given precondition is guaranteed to terminate in a state satisfying a given postcondition). While writing OCP, two (of several) additional formalization issues I encountered for the derivation of concurrent programs were:

- Just having assertions associated with control points in a program suffices for specifying safety properties ("bad things don't happen") but not for specifying liveness properties ("good things do happen").[2] So the problem formulation—program skeletons with assertions—that Dijkstra used for his derivations could not handle important aspects of the required behavior for a concurrent program. Termination, entry into a critical section, and eventual collection of an unreachable node are all examples of liveness properties.

[2] Lamport [1977] had introduced this classification of concurrent program specification into safety properties and liveness properties, and he had proposed methods for verification of both.

- To be formal, rules are needed for deleting ghost variables. Dijkstra had not given them, and the rules are not straightforward. A program with ghost variables has a different state space, a different set of atomic actions, and (potentially) a different set of control points. Properties that are sensitive to those program attributes thus could be affected by deleting the ghost variable. Moreover, ghost variable deletion can be unsound because a control point that was not being reached before deletion of an assignment to the ghost variable might be reached after deletion of that statement [McCurley 1989].[3]

When I embarked on writing OCP, I was expecting to describe an existing body of work—not to develop new apparatus for reasoning. The above problems (and various others) with formalizing the use of interference-freedom for the derivation of concurrent programs were a surprise. I often would discover a problem only because I was trying to make OCP be self-contained so that readers would be equipped to follow Whitehead's dictum. My decision that OCP would describe a method whose uses were fully formalizable—thus allowing uninterpreted manipulation to replace thinking—meant that before talking about concurrency OCP had to devote 130 pages to the description of a temporal logic (so that both safety and liveness could be handled) and a predicate logic that enabled assertions to characterize program state (including the program counter) and execution history, so ghost variable deletion can be put onto a sound footing. That prefix of 130 pages is covering ground that Dijkstra and his school hadn't.

The approach advocated in OCP has not been widely embraced by programmers or by instructors who teach concurrency. But the approach that *Discipline* advocated for deriving sequential programs also has not been widely embraced by programmers or instructors. We can only speculate about the reasons. High on the list is likely to be the requirement that practitioners of these approaches must be facile with manipulating predicate logic formulas. Entering college students in the United States have learned to manipulate algebraic formulas and have gotten plenty of practice. The same cannot be said about their facility with manipulation of predicate logic formulas; uninterpreted formal manipulation of assertions and invariants is a challenge for these students. Operations these students perform "without thinking about them" are more difficult and offer less assurance in the result than operations they perform by thinking. So evoking Whitehead's dictum does not deliver the promised benefits. That lack of facility with

[3] This problem arises if the assignment statement involves an expression that is undefined in certain states and, therefore, the assignment statement does not always terminate.

predicate logic prompted Gries and me to write a sophomore-level textbook [Gries and Schneider 1993] that would give students the skills and the confidence to perform uninterpreted manipulation of predicate logic formulas "without thinking about them."

27.2 Dijkstra's Influence: On Becoming a Researcher

I had read Dijkstra's published papers [Dijkstra 1968b] on the THE multiprogramming system and on concurrent programming [Dijkstra 1968a, 1972] in a seminar that dissected seminal papers in Operating Systems. Only after I got to Cornell in Fall 1978 did I learn about Dijkstra's unpublished EWD-series of short technical reports. From time to time, Gries would receive a batch of these EWDs in the mail, which he circulated to the faculty in our department.

EWDs were typed by Dijkstra, photocopied by him, and mailed to a set of his acquaintances, who were expected to serve as the internal nodes in a dissemination tree. EWDs were not like the computer science papers that I had been reading (and I had believed that I should be writing). Some EWDs were technical and solved small problems, focusing not on the solution but instead on how the solution was obtained. Other EWDs were philosophical musings about the field and about other computer scientists. Still others gave travelogues about Dijkstra's participation in meetings and visits to other institutions. The nontechnical EWDs often voiced strong opinions about research culture and about research he had been shown. Dijkstra did not hesitate to say that he thought a question being studied was ill-chosen, that the solution being advocated was too complicated, or that a presentation had been flawed. There also were EWDs that questioned long-accepted notational conventions, including the style and formatting of mathematical formulas and proofs.

Reading EWDs inspired me to ask different research questions from what I was seeing in the Operating Systems and Software Engineering communities. As a result, my research focus changed to questions about methodology, to the design of new abstractions, and to the justification of foundational definitions. But I also was terrified to be reading about norms that, apparently, Dijkstra (and I imagined other top computer scientists) would be applying to my lectures, my writings, and my research. As an assistant professor, I was trying to make a place in the community, and I believed that the opinions of leading researchers would play a big role in how I was going to be perceived. I was also now learning from Dijkstra's EWDs that practices I used were objectionable. The use of anthropomorphism or diagrams in lectures, for example, had seemed to me natural for delivering explanations of technical material, yet Dijkstra proclaimed that such explanations would confuse.

After my first year at Cornell, the author behind those EWDs became a flesh and blood person in my life. It was August 1979, and University of California at Santa Cruz was running a three-week series of public lectures on programming methodology. Back then, programming methodology promised to provide a foundation that would enable us to produce software that satisfied its requirements. Week 1 of the lecture series would be a tutorial about formal derivation of sequential programs. Weeks 2 and 3 would comprise lectures delivered by members of IFIP Working Group 2.3, a group of leading researchers in programming methodology.

Gries had originally been scheduled to deliver the tutorial, assisted by his Ph.D. student Gary Levin and by me. That tutorial was based on a semester-long course Gries had been teaching to Cornell's first-year Ph.D. students; it was an explication of the contents of *Discipline*. Registration for the tutorial exceeded expectations. The organizers responded by adding a parallel section, and they hired Dijkstra, along with his assistant Wim Feijen, to teach it.

So that August, I found myself in Santa Cruz, meeting each night with Dijkstra, Feijen, Gries, and Levin to decide what should be covered in class the next day. That decision rarely took much time, since Dijkstra and Gries had long ago converged on a small set of example program derivations for illustrating the various strategies for devising a loop invariant to compute some specified result. So, with little else to do on a summer evening, our meetings would drift into exploring small mathematical problems and programming problems. No surprise: in these discussions, the method for obtaining a solution was far more important than the solution itself. And I got to see how Dijkstra would think about these problems. Besides these evening meetings, the teaching staff (and accompanying family members) would typically sit together at meals in a student cafeteria. All that contact provided opportunities for me to talk with Dijkstra about my own research and to get his feedback. I also got to know his wife Ria and his daughter Femke, who also had come to Santa Cruz.

Gries didn't mind if I attended Dijkstra's lectures. But I was in Gries's lecture when the magnitude 5.7 Coyote Lake 1979 earthquake (my first) struck at 10:05 local time on Day 1 of our course—Gries blamed the event on something Dijkstra had just said. When I did attend Dijkstra's lectures, they were quite different from classes I had taken or taught. I have a record of my reactions in journal entries that I was writing each night. Here is the list of points I recorded about Dijkstra's lecturing style.

- Start each lecture by having a notable quotation on the board.
- Pause long enough so that you can think through the entire next sentence before you start speaking.

- Say things once and precisely rather than giving approximations in multiple ways.
- Don't be afraid to be philosophical or to pontificate.
- Prefer to write on the blackboard over using preprepared transparencies.
- Lecture without notes; derive everything as you need it.

You can imagine how frustrating these idiosyncrasies could be for some of the audience members. The long pauses between Dijkstra's sentences made it easy to get distracted. And because Dijkstra would say something only one way and only once, a listener whose attention even momentarily was elsewhere (say, because something said earlier was puzzling) could miss important content. Needless to say, opinions about Dijkstra's lecturing style varied.

For Dijkstra, giving a lecture was "delivering a performance" (his words)—not unlike a concert. He aimed to create suspense (with the pauses), and he aimed to end on a surprising note, preferably having derived something beautiful. I found inspiration in this, and I adopted some of his lecturing idiosyncrasies. For example, after that summer I have used the blackboard rather than transparencies whenever I have taught classes.[4] (I do use slides when I speak at a colloquium or conference, because it's the only way I am able to cover enough material.) Also, for a time, I would start my lectures with one or another of the quotations from Dijkstra's collection. Gary Levin and I had compiled a list.

An invitation to stay with Edsger and Ria Dijkstra the following March (1980) at their home in Nuenen was my next chance to spend time with the Dijkstras. I still find it amazingly generous that I, then only a casual acquaintance, was invited to stay at their home. Apparently, it was not unusual for them to host scientific visitors. My visit to Nuenen was the final stop on my first visit to Europe, so it was broadening in all kinds of ways. I found life at the Dijkstras' to be what one imagined for the life of an academic (but nothing like my life as a US academic). Quoting from my journal: "Breakfast, followed by tea in the living room. Sherry at 5pm. Piano playing after dinner, with tea. A fire in the fireplace." I was not surprised by the heavy brown bread (my first stop on this trip had been Munich), but I was surprised by the Dutch habit of putting chocolate sprinkles on toast and by Dutch adults at meals drinking milk. Like other visiting Computer Scientists to the

[4] These days, I tell students in my classes who wish I was using and distributing copies of transparencies to use their phones and photograph the blackboards after the lecture. Whatever would have been on the slides is still there on the blackboard, along with connections—due to proximity and added arrows—that are not easily conveyed on slides.

household (as I was told), Dijkstra and I toured Nuenen on his tandem. I got to see an operating windmill as part of that tour.

The official part of my visit was spending two days visiting the Technical University of Eindhoven (THE). I was a guest lecturer in Dijkstra's Tuesday morning class on Synchronization (24 students, at the level of Cornell seniors). I also was a guest of the Tuesday Afternoon Club (which had 10 people, including several whose names I recognized from having read EWDs). According to my journal entry, both lectures were well received by Dijkstra. In the morning class, I described serializability in databases and included a new distributed implementation of semaphores; in the afternoon, I presented a general approach to distributed synchronization. I tried to display my best Dijkstra-form for these lectures. I started each of the lectures with a quotation (though not one of his). And I delivered both lectures using the blackboard.

Over the following years, I would see the Dijkstras from time to time. They periodically visited Gries in Ithaca. I became a member of IFIP WG2.3, so I would often see Edsger and Ria at those meetings, which took place all over the world. Dijkstra's research moved away from a focus on programming to a focus on a calculational predicate logic with a new "everywhere" operator. I too turned to calculational predicate logic, working with Gries to give a calculational presentation of discrete math for sophomores—but without "everywhere" which we sorted out only later and discussed in our paper [Gries and Schneider 1998].

It has been 20 years since Edsger Dijkstra died. But he remains ever present in my head. I hear him delivering a rebuke whenever I or somebody else uses anthropomorphism (suggest: "suppose i is a processor"), writes a formula involving subscripted subscripts, or when the proof involves pulling a rabbit out of a hat or proceeds in a circuitous way ("Beauty is our business."). And the Dijkstras are also very much present in our kitchen, represented by a pair of reversible two-color double-knit potholders that Ria made for us as a house gift.

Acknowledgements

David Gries and Leslie Lamport provided helpful comments on an early draft of this chapter. Schneider is supported, in part, by AFOSR grant F9550-19-1-0264.

References

E. W. Dijkstra. 1968a. Cooperating sequential processes. In F. Genuys (Ed.), *Programming Languages: NATO Advanced Study Institute*. Academic Press, 43–112.

E. W. Dijkstra. 1968b. The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5, 341–346. DOI: https://doi.org/10.1145/363095.363143.

E. W. Dijkstra. 1968c. Go to statement considered harmful. *Commun. ACM* 11, 3, 147–148. Letter to the Editor. DOI: https://doi.org/10.1145/362929.362947.

E. W. Dijkstra. 1972. Hierarchical ordering of sequential processes. In C. A. R. Hoare and R. H. Perrot (Eds.), *Operating Systems Techniques*. Academic Press, 72–93.

E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

E. W. Dijkstra. 1979. A tutorial on the split binary semaphore. EWD703. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD703.PDF.

E. W. Dijkstra. 1980. The superfluity of the general semaphore. EWD734. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD734.PDF.

E. W. Dijkstra. 1982. A personal summary of the Gries–Owicki theory. EWD554. In E. W. Dijkstra (Ed.), *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, New York, 188–199. DOI: https://doi.org/10.1007/978-1-4612-5695-3_33.

E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11, 966–975. DOI: https://doi.org/10.1145/359642.359655.

R. W. Floyd. 1967. Assigning meanings to programs. In J. Schwartz (Ed.), *Proc. Symp. on Mathematical Aspects of Computer Science*. American Mathematical Society, 19–32.

D. Gries. 1981. *The Science of Programming*. Springer Verlag, New York. DOI: https://doi.org/10.1007/978-1-4612-5983-1.

D. Gries and F. B. Schneider. 1993. *A Logical Approach to Discrete Math*. Springer Verlag, New York. DOI: https://doi.org/10.1007/978-1-4757-3837-7.

D. Gries and F. B. Schneider. 1998. Adding the everywhere operator to propositional logic. *J. Logic Comput.* 8, 1, 119–129. DOI: https://doi.org/10.1093/logcom/8.1.119.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10, 576–580. DOI: https://doi.org/10.1145/363235.363259.

L. Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* 3, 2, 125–143. DOI: https://doi.org/10.1109/TSE.1977.229904.

E. R. McCurley. 1989. Auxiliary variables in partial correctness programming logics. *Inf. Process. Lett.* 33, 3, 131–133. DOI: https://doi.org/10.1016/0020-0190(89)90191-9.

S. Owicki and D. Gries. 1976. An axiomatic proof technique for parallel programs. *Acta Inform.* 6, 319–340. DOI: https://doi.org/10.1007/BF00268134.

F. B. Schneider. 1997. *On Concurrent Programming*. Springer-Verlag. DOI: https://doi.org/10.1007/978-1-4612-1830-2.

A. N. Whitehead. 1958. *An Introduction to Mathematics*. Oxford University Press.

28 Edsger W. Dijkstra in the Eyes of His Friends, Colleagues, and Students

28.1 Lex Bijlsma, Open University of the Netherlands

Having been trained as a number theorist, I had absorbed the prevailing attitude among my fellow researchers, namely that a problem is solved when only finitely many cases are left to check. Most theorems in this area simply state that there exists a computable number C such that all integers greater than C satisfy some hypothesis. This is supposed to solve the problem, as the reader is invited to check the numbers up to C by hand. Never mind that in the few cases where someone actually took the trouble to compute C, it turned out to exceed the number of particles in the universe: finite domains were considered to be trivial on principle.

I suspect that Edsger's invitation to attend his weekly Tuesday Afternoon Club was motivated by his curiosity as to how a working mathematician would look at problems in computing. It is not inconceivable that my reactions contributed to the way mathematicians are described in "On a cultural gap" (EWD913). But once I had overcome my prejudices, I became intrigued, not so much by the subject itself as by the disciplined thought habits cultivated in this circle and by the astounding efficacy with which these could tackle quite challenging problems.

One thing I learned to do differently is the presentation of proofs. The mathematicians' style is to make these short, efficient, and as dazzling as possible. The result is usually a proof that begins with the introduction, out of thin air, of some complicated function that will turn out to provide just what is needed. Such proofs give no indication of the way they were constructed, thus obscuring the underlying design decisions and possible opportunities for improvement. Whenever encountering a proof starting with a completely unmotivated definition, Edsger would call out the word "rabbit", referring to the animal old-fashioned stage magicians used to pull out of their top hats.

The way Edsger preferred to organize proofs was in terms of “hint calculus”, a calculational style (developed in cooperation with Wim Feijen) where every step is annotated with an argument for its suitability. The foundation for such proofs is a particular style of predicate calculus[1] that owes more to the mathematics of lattices than to logic. His initial disregard of the established concepts of logic caused some friction between the group around Edsger and that of Eindhoven mathematician Dick de Bruijn. An attempt to heal the rift by showing the connection between the two approaches to logic was later undertaken by Rob Nederpelt from De Bruijn’s entourage and myself[2].

Another habit I picked up is looking at formulas as objects of interest in themselves, not just as shorthand for vaguely imagined entities floating in Banach space or wherever. This requires paying a lot of attention to notations that are amenable to efficient manipulation but provides the advantage of a medium where reasoning can take place outside your head. Consequently, much greater complexity can be tackled: think of the way multiplication on paper lets you deal with much larger numbers than mental arithmetic allows.

A famous quote of Dijkstra is the following: “I mean, if 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself ‘Dijkstra would not have liked this’, well, that would be enough immortality for me” (EWD1213). I can testify that this actually works. I cannot introduce a notation without wondering whether all the subscripts and parentheses cannot be eliminated and whether the size and the symmetries of an infix operator symbol accurately reflect the properties of the abstraction it signifies. And it took me several minutes to decide if the introduction of identifier C in the first paragraph was justified.

28.2 Ken Calvert, University of Kentucky, Lexington, KY

When I arrived in Austin to begin Ph.D. studies in the summer of 1984, Professor Dijkstra had been at the University of Texas (UT) for only a year or so, but the legends around him had already started to grow. It was said amongst the graduate students that his office, on the 21st floor of the Texas Tower, housed a grand piano that had to be hoisted by crane up the outside of the building because the elevator was not large enough for it. The students knew—and this was no myth—that his was not an ordinary class, including as it did the prospect of a one-on-one *oral* final examination with a Turing Award winner.

[1] E. W. Dijkstra and C. S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer.

[2] L. Bijlsma and R. Nederpelt. 1998. Dijkstra–Scholten predicate calculus: Concepts and misconceptions. *Acta Inform*. 35, 1007–1036. DOI: https://doi.org/10.1007/s002360050150.

I knew of Dijkstra's shortest path algorithm and the semaphore abstraction from my prior studies, and his presence on the faculty was one of the reasons I chose UT for my doctoral studies. However, I was not familiar with his work on programming methodology. One day that summer, as I was browsing in the UT bookstore, I happened across *A Discipline of Programming* (probably in a display of books by "local authors") and bought it. It changed the way I think about programming forever. I had considered myself a pretty good programmer and had taken at least one computing theory course in my prior studies, but somehow it had never "clicked" for me that *real* programs were amenable to formal mathematical reasoning and even derivation. That may have been the first of the many things I learned from Professor Dijkstra (henceforth EWD), who had a great influence on me during that formative period of my career through his classes, his writings, and the people around him.

It took me a year or two to muster the courage to take his *Capita Selecta* course, and when I did I took it "pass–fail". (For those not familiar with the American educational system, that is a lower-stress option compared to the normal system, in that it does not distinguish among levels of performance above the threshold required to receive credit for the course.) When I was subsequently invited to join the Austin Tuesday Afternoon Club (the group of students, faculty, and visitors who met in EWD's office weekly), I was both thrilled and intimidated. I learned then that—by that time anyway—there was no grand piano in EWD's office.

Meetings of his course almost always featured a quote of some kind, already written on the blackboard—EWD did not care for "whiteboards"—when we arrived. These quotes dealt with topics ranging from music to general words of wisdom, to things that tickled his sense of humor, and, of course, computing science. Some examples from the Fall of 1987 and Spring of 1988 included:

- "He is the master of us all."—*Hayden on Handel after hearing the Halleluja Chorus*
- "Learn to write well, or not to write at all."—*John Dryden*
- "Elevators of America"—*name of a company making elevators*
- "My mother loved flowers and that's how I got started"—*Around Austin*, 1988.04.12.
- "... enter program maintenance into the very mainstream of software engineering."—*Colin Stewart, CACM* Sept '88
- "However, both companies expect an upturn [of the market] once AI products are shown to work."—*Computer*

These quotes, along with off-the-cuff verbal comments in classes and other public fora, communicated EWD's opinions and values—what he found interesting or ridiculous, his views on the computing field, the world, and the local community—and did so in a subtle but powerful way. All teachers convey values to their students, implicitly or explicitly, but before I met EWD I had never encountered someone with such strongly held convictions who also had the intellect and confidence to share and defend them. In my opinion, this openness of his personality magnified his technical influence and contributed to the "larger than life" impression he made on so many people—for better or worse.

Meetings of the ATAC were typically devoted to reading a manuscript written or suggested by one of the members or EWD himself—often a new addition to his "EWD" series of notes. Occasionally, the session was devoted to discussing potential Ph.D. thesis topics. I learned much about how to think, speak, and write precisely from Professor Dijkstra and those in his orbit—and also about the importance of doing so. Among the principles, wisdom, and values I got from those experiences, Rule 0 ("Don't make a mess of it") is one that all my students, not to mention my children, are required to learn. Occasionally, when it seems especially appropriate, I have my class recite it together. Among other pearls I picked up that are still with me, I would mention these:

- Notation matters. Some "standard" mathematical notations are not only not helpful but actually hinder understanding. An hour spent refining notation to save a hundred readers a minute of thinking is well spent.
- The purpose of a calculus is to *let the symbols do the work*—to the greatest extent possible. I still use the calculational proof style I learned in EWD's classes and presented in *Predicate Calculus and Program Semantics*.
- "If you must choose between beauty and utility, choose beauty, because the world has enough ugly useful things." (The last part is at least as accurate now as it was then.)
- The importance of being able to understand a new concept or abstraction through its properties ("axiomatically"), rather than by analogy. From my notes in EWD's class September 17, 1987: "Patterns of understanding of novelty: only one generally taught is analogy. If this is all you have, you are doomed when it comes to understanding ideas so radical that every analogy by definition is too shallow.... The only way to understand such things is to try to *avoid* analogy... Understand the new thing on its own terms."

Observing EWD's interactions with the brilliant and famous people who visited Texas and sat in his class or the ATAC was always instructive. Sometimes those interactions were cordial, sometimes not. I recall him sitting in the back of the room during a talk by one distinguished visitor and making rude noises.

Dijkstra was helpful to me in my career, enabling me to attend a NATO Summer School in Marktoberdorf, serving as a reference in my job search, and staying in touch for quite a few years. He was kind and generous to me and my family personally while I was still a student, stopping for a beer and a visit at our campsite when we chanced to meet in one of Texas' state parks (he and Ria were camping in their VW "Touring Machine"); inviting my wife and me to dine in his home as I neared completion of my studies. On that occasion, I expressed admiration for the design of a Brabantia corkscrew he was using. He said, "Wait a bit," rummaged in a cabinet for a few moments, pulled out one just like it—brand new and still in its package—and presented it to me. Like so many things great and small that I received from EWD, more than 30 years later I am still using that corkscrew. Scarcely a week goes by when I don't quote something he said or refer to something I learned from his class or from the ATAC. I'm pretty sure I have not always done those things justice, but I hope that they have, in some small way, been reflected to those in my own circle of influence over the years.

28.3 K. Mani Chandy, California Institute of Technology, Pasadena, CA

I was the Chair of the Computer Sciences Department at the University of Texas at Austin when the university made an offer to Edsger to join the department. Many had misgivings. Everybody agreed that Edsger was a great computer scientist, a genius. Everybody also agreed that he was among the most outspoken, blunt, opinionated scientists on the planet. Moreover, Edsger was proud to belong to that group. Edsger was particularly skeptical about Artificial Intelligence and was known to voice that skepticism. The CS department at UT was a broad church. And broad churches survive on tolerance while Edsger was notoriously intolerant. Nevertheless, we made Edsger an offer and I think it was one of the best decisions that the department made. The decision was not without controversy then and remains controversial even now.

Edsger influenced many people in the department. The Austin Tuesday Afternoon Club (ATAC) meetings were a delight. His book with Carel Scholten, *Predicate Calculus and Program Semantics*, taught us the beauty of calculational proofs. Edsger required that we read papers aloud at the ATAC. This seemed to me to be an old-fashioned tradition from days when scholars actually "read papers" as opposed to modern presentations of colorful slides and videos. I learned, however, that reading aloud exposed redundancies, unnecessary adjectives, and shoddy writing. Flashy presentations hid all of that. I still read my papers aloud to myself.

Figure 28.1 Professor Dijkstra and the author at the Calvert home, sometime in late 1996 or early 1997.

As the chairman of Edsger's department, I sometimes found him painfully blunt. Not for him the gentle reprimand. He also had a wicked sense of humor. For example:

- He asked my wife and me on hearing that she had a Ph.D. in psychology: "How do you two manage to stay married?" Edsger must have felt that psychologist—CS relationships were unstable.
- His words of encouragement when he found out that I graduated from MIT: "I am happy that you seem to be working at overcoming your poor education!"
- While I was giving a chalkboard talk, he announced: "I've never met an American computer scientist who could give a talk without drawing two boxes and joining them by a line!"

Edsger and Ria used to drop in unannounced at our house for a chat and a drink, a nice habit common in India but less so in the US. My three-year old son, getting ready for his bath and bedtime story, came downstairs in the nude, saw Edsger and said—much to Edsger's delight—"Oh no! Not you again!"

I often had Edsger over for dinner with students, and he was always entertaining and pleasant, belying his reputation of intellectual haughtiness. Edsger was extremely kind, a kindness he hid in his gruff exterior. For example, he was friendly to my parents, apparently enjoying their conversation, though my father and mother couldn't have been more different than Edsger and Ria.

Jayadev Misra and I had lunches with him regularly at the Santa Rita room at UT, and these were wonderful, happy times. He was often funny with Oscar Wilde-like quips. He also educated the two of us, over these lunches, about programming: what it meant to him and what it ought to mean to us. Edsger was very helpful to us, reading our papers, making helpful suggestions: encouraging while remaining a Dutch Uncle. When I write papers, even now, I still see Edsger over my shoulder going "Tsk! Tsk!"

Here's a little example of how Edsger influenced us. We used implication, $P \Rightarrow Q$ as a predicate at times and as a Boolean at other times, expecting our audience to understand the meaning by context. And the audience almost always got the intended meaning. Edsger, however, thought that was sloppy. He and Scholten introduced the square bracket to identify Booleans, as in $[P \Rightarrow Q]$. Just a little notational thing, but an important one. I found that being less sloppy not only helped my readers understand what I had written, but most importantly, helped me from confusing myself.

Edsger was *the* major influence on Jayadev Misra and me as we wrote our book on concurrent computing. Edsger's impact shows in our calculational proofs and the emphasis, throughout the book, on formalism and logic. If, instead, we had used anthropomorphisms—which are quicker to write, and perhaps more readable—we would have finished sooner and perhaps had a wider audience. But we have never regretted the Dijkstra-effect and remain forever grateful.

28.4 Eric C.R. Hehner, Department of Computer Science, University of Toronto

I have had a career-long interest in formal methods of program design, and I still teach a course with that title. My introduction to the subject was the book *A Discipline of Programming* by Edsger W. Dijkstra in 1976. I wrote my first paper on the subject that same year. So, I was eager to meet Edsger when he came to Toronto for the IFIP Congress in 1977. He had read my paper and agreed to meet me. He introduced his just completed Ph.D. student Martin Rem and me to each other. Martin and I connected both personally and in our research interests, freeing Edsger from babysitting duties.

Figure 28.2 Elaine Gries, Eric Hehner, and Edsger Dijkstra during Dijkstra's 60th birthday party, Austin, Texas, May 1990.

Following the IFIP Congress, Edsger went to an IFIP Working Group 2.3 meeting in Niagara-on-the-Lake, where I was the newly appointed secretary of the group. My paper, which was critical of some aspects of Edsger's work, was a topic of discussion. Edsger was a very important person, and I was nobody, and that provoked several people to defend Edsger. But Edsger found merit in my criticisms. He was more willing to consider how his own work could benefit from the criticisms than some of the others in the room.

After the working group meeting, Edsger had a couple of days to kill before his flight back to the Netherlands, so he came to my house. My mother was also visiting just then. Back then, my mother and Edsger were smokers, and I didn't allow smoking in the house, so they went out on the front porch together. My mother happened to mention that she was not fond of existential proofs with no instantiating example (witness). Well, after that, Edsger and my mother talked only to each other; I might as well have been dead.

Four years later I had the opportunity to return the visit, staying in Nuenen. Since their son Rutger was away, I was given his bed. As we were heading out for a ride on the tandem bicycle, Edsger's wife Ria noticed a hole in the back of my pants. Edsger tried to shush her, but too late. He had noticed it too, but he didn't know if I had a replacement pair. He reasoned that I would be better not knowing there was a hole than knowing and unable to do anything about it. Fortunately, I did have another pair. But it stuck in my mind as an example of Edsger's kind consideration, quite opposite to his stated philosophy of "telling truths that hurt".

After Edsger moved to Austin, he held the Year of Programming in 1986–1987. A variety of formal methods researchers spent all or part of the year in Austin, and I was fortunate to be included. This put me in almost daily contact with Edsger for several months. One of the benefits for me was sitting in on Edsger's undergraduate course on Mathematical Methodology. I sat with Wim Hesselink because we were somewhat out of our age group. Each class Edsger arrived with a problem to be solved. The lesson was how the formal expression of the problem guides the solution, or in his words, "let the symbols do the work". What I learned in that class permeates my formal methods course today.

Another benefit to me of the Year of Programming was participation in the Tuesday Afternoon Club. I had attended one or two of these in Eindhoven, and in Austin I attended several more. Each week we read a paper, chosen by Edsger; someone read aloud so we were synchronized and the pace was slow. Anyone could ask a question or make a comment or criticism at any time, and the criticism could be at any level, from syntactic (how well is the idea expressed) to semantic (how valid is the idea) to judgmental (how important is the idea). I was amazed at how productive and effective this format is. On my return to Toronto, I instituted a copy of the Tuesday Afternoon Club (but not on Tuesday afternoon).

From 1977 to 2001, about every 9 months, I had the pleasure of talking and dining with Edsger at IFIP Working Group 2.3 meetings. At one such meeting one evening, Edsger saw that I was stuck listening to a known pompous bore. Edsger came across the room, apologized for butting in, and said that I was urgently needed to settle an argument in his group. He rescued me.

Toward the end of his life, Edsger became concerned about his legacy. One day, walking to lunch, Edsger said to me "In a hundred years, the only thing left of all my work will be the shortest path algorithm.". He may have expressed the same thought to others because J Moore said, In Memoriam, "Without a doubt, a hundred years from now every computer scientist will study Dijkstra's ideas, including

- the mathematical basis of program construction,
- operating systems as synchronized sequential processes, and
- the disciplined control of nondeterminacy,

to name but three.". In my last letter to Edsger, I said "All of us have taught literally thousands of students, who take away thoughts that originated in your head. And your influence is not just technical. Your way of working, and your ethics, have become mine as well as I am able to emulate them. What I want to say is: thank you. With great admiration and affection, Rick". His last letter to me, handwritten of course, dated Nuenen, Friday, July 19, 2002, just 18 days before he died, said "I thank you for your friendship.". I treasure that letter.

28.5 Wim H. Hesselink, University of Groningen

The first time I met Dijkstra was on Tuesday, July 16, 1985. This meeting came about as follows. I had got my doctorate in pure mathematics from the University of Utrecht in 1975. The next year I moved to Groningen. In 1983, I had lost sight of the research front in my branch of mathematics. Around the same time, Jan van de Snepscheut, a former student of Dijkstra, was appointed professor in computing science in Groningen. He turned out to be an inspiring leader of an emerging research group. This gave me confidence to move to computer science. To finalize this move, the Institute granted me a sabbatical year with Dijkstra at the University of Texas at Austin, where Dijkstra had been appointed in 1984.

The meeting with Dijkstra took place in Nuenen. He had announced to have a beard and sandals, and a VW Sirocco. I had not expected him to wear shorts. He spoke softly. I often was thinking of raising a new topic when he would proceed with the previous one. Jan had prepared me for this by admitting he found it difficult to talk to Dijkstra over the phone. After explaining my switch to computer science, I started with the remark that my main difficulty in the new field was to decide which things were important. Dijkstra seemed to approve of this, and told an anecdote about certain logicians who, confronted with linear search, asked about the computability of the search criterion. He explained that the central problem of computing science was to restrain the complexity of our artefacts, and not to make a mess of it. I tried in vain to seduce him to divide computing science into subdisciplines, as I knew mathematics could be divided.

From September 1986 until May 1987, I worked in the second room of Dijkstra's office in Austin. My weekly highlight was the Austin Tuesday Afternoon Club (ATAC), where Edsger always could inspire a small group of colleagues to investigate and discuss some problem. The first problem was a preprint by Greg Nelson about Dijkstra's calculus. We concluded this investigation in two or three sessions and found some minor mistakes. Edsger strode to the phone and informed the author about our findings. This way of compromising the anonymous reviewing system was new for me. My main learning experience was not in the contents of Nelson's paper but in the way Dijkstra and the others appreciated it. At the end of my year in Austin, there was an ATAC session in which Pnueli was invited to explain his temporal logic. He had difficulty in defending the relevance of his logic against the skeptical attacks of Dijkstra and Hoare. Indeed, Hoare was that year also in Austin, and often attended the ATAC. This was helpful because Dijkstra alone could have been overwhelming.

Next to the ATAC, I attended an undergraduate honors course Edsger gave on mathematical methodology. Quite strange, since I had a doctorate in math, and knew it much better than Edsger. Yet I appreciated this course and had the impression I learned things. Once in the course, I used hands while explaining something. Edsger then told me to try to do it without hands. Some weeks later, I saw Edsger using hands to explain the lexical order, but I kept this to myself. During this year, I worked on several ideas suggested by Dijkstra or Hoare. Invariably, Dijkstra's suggestions turned out mathematical, while Hoare's ideas involved languages.

The sabbatical year with Dijkstra has broadened my view of computer science immensely, because of the contact with Dijkstra, Hoare, and other colleagues, and because of the string of conferences of the Year of Programming that was organized in Austin. The direct influence of Dijkstra was limited. Before coming to Austin, I had read and used EWD883 (by Dijkstra and Scholten), but when I told Edsger this, he said he did not like it, presumably because it was too operational. Another influence was that afterwards I have organized in Groningen, for 20 years, a reading group, with some staff and Master's level students, as a weak simulation of the ATAC.

In subsequent years, Edsger and I mainly corresponded by letters. In 1988, I attended the Summer School at Marktoberdorf, with Dijkstra as one of the directors. Between the sessions he was often surrounded by a group of disciples. In 1989, he attended the first MPC conference in Twente, which was organized by our Institute. In March 1991, I suggested to him in a letter to collaborate on something like concurrency. He declined gracefully. About concurrency, he wrote: "I don't like it, because it is such an operational concept". In the winter of 1994, we were all deeply shocked by the tragic death in California of Edsger's beloved disciple Jan van de Snepscheut. Edsger and I extensively communicated about Jan's history in Groningen.

Edsger encouraged his followers to imitate his behavior. I was too old to comply, but I had a beard when we first met, and my handwriting was acceptable though quite different from his own. Edsger respected my independent intelligence, and we shared the love for a convincing mathematical argument. Anyhow, following Dijkstra with his EWD series, I started to enumerate my manuscripts as whhxxx. (This one is whh572.) I still do this, though not as rigidly as Edsger would have wanted. This is one of Dijkstra's lasting influences. In spring 1987, around the time of EWD1005, Luca Cardelli had distributed a pamphlet marked EWD1024 in a font that emulated Edsger's handwriting. Edsger and Ria were not amused. Eleven years after his death in 2002, I met a brother of Edsger. When I told him I had been with Edsger for a year, he looked at my feet and asked: "Why don't you wear sandals?"

28.6 Rajeev Joshi, Amazon, Pasadena, CA

I first met Edsger when I took his course, *Capita Selecta: Selected Topics in Computing Science*, at the University of Texas at Austin. The course description stated that the "course is not about mathematical results but about doing mathematics" (EWD1220). When faced with a conjecture of the form *given P, show Q*, Edsger would start by questioning whether each assumption in *P* was really needed. This exercise gave a deeper understanding of the problem and could point the way to the solution.

He had a distinctive way of speaking. He never used filler words like ums and ahs and you-knows. But he would often pause in the middle of a sentence, gaze into the distance, and then continue a few seconds later. His sentences were always complete and grammatically correct. It was clear that he loved to teach. Each time he finished a proof, he would stand back delightedly, and say "Ain't it a beauty!". He enjoyed interacting with students, always encouraging us to share our ideas, gently pointing out mistakes in proofs, but never making a disparaging remark. He did, however, warn us in the very first lecture: "Don't compete with me: firstly, I have more experience, and secondly, I have chosen the weapons."

The course grade was based on a private oral examination in his office. I remember walking into the office, understandably nervous. But we started with small chat and a cup of coffee (in a mug with the slogan, *Rule 0: Don't make a mess of it*, of which I now have a copy). That settled me, and minutes later, we had started a journey exploring graphs of functions. By the end of the exam, we had derived two well-known results from number theory (see EWD740 and EWD742, extensively discussed in Chapter 12 written by Jay Misra). Along the way, Edsger played the role of guide, gently nudging when necessary, but mostly letting me find the proofs for myself. It was easily the most unusual, but also most exhilarating, exam I had ever taken.

After the course, I started attending the weekly meetings of the Austin Tuesday Afternoon Club (ATAC). We would usually read a paper, often taking turns reading aloud, going over the material with a fine-toothed comb. He was extremely picky about notation; he argued that poor notational choices often made it harder to think about the problem, and always harder to read the solution. Even the size of a symbol mattered (higher precedence ones should be smaller). He often said it was the duty of a writer to avoid wasting the readers' time with sloppy or obscure writing. I learnt this the hard way when I had submitted my thesis for review. One day, Jay Misra mentioned that Edsger had found a mistake in one of my proofs. Horrified, I spent the entire evening going over every single proof but not finding anything. The next day, Edsger accosted me in the hallway. He said he had finally been convinced that the proof was correct, but the hint I had provided for one step was so clumsily worded that he had been forced to work out the proof himself, and he was most annoyed with me. In spite of the scolding, I was gratified and humbled by the effort he was clearly putting into reviewing the thesis.

Through ATAC meetings, I came to know Edsger the person. I learnt that, when he was a student at the University, at the end of each day he would go for a walk and recreate from memory every lecture he had heard that day. He said this made clear to him what he hadn't fully understood. It was another example of how disciplined he was in his approach to everything in life. On another instance, while visiting his home and seeing a photograph of him as a young man, I commented on how handsome he looked in the photograph. With a twinkle in his eye, he said, "but I am still handsome".

Edsger advised no graduate students, yet he profoundly shaped the thinking of so many of us. His uncompromising integrity had created an unfortunate public perception of him as gadfly, as someone who was arrogant and abrasive. But in personal interactions, he was remarkably generous and thoughtful, especially with students. In EWD561, writing about how he prioritized his time, he wrote: "A self-imposed constraint is that all appeals to my assistance in which the professional life of others is at stake are dealt with promptly". I had direct experience of this when I once hesitatingly asked if he would write a letter in support of my application to the Marktoberdorf Summer School. He immediately invited me into his office, sat down at his desk, and started writing. Two minutes later, he handed me a letter and told me to mail it to the organizers. (Of course, I sent a copy, and kept the original!)

Edsger's great genius was his ability to strip away the inessential, leaving behind only what is absolutely needed. One striking example of this was his explanation of the Chandy–Lamport Distributed Snapshot algorithm (EWD864a) using colored messages. Once you read his explanation, it's immediately clear how the algorithm works, why it's correct, and—best of all—the explanation is so elegant that you can never forget it.

A second striking example, one that directly influenced my work, is EWD1225, *Misra's weakest fair buffer*, which implements an unordered buffer that is also fair. Fairness here refers to the property that, given an infinite sequence of insertions and removals, every value inserted will eventually be removed. The ingenuity in Misra's construction is that the buffer is also *weakest* in the following sense: it has sufficient internal nondeterminism that it is capable of generating *every* possible fair reordering. Proving that the buffer is fair is straightforward, but showing that it is weakest is nontrivial. In EWD1225, Dijkstra showed an elegant proof of the latter property using auxiliary variables. Given any fair reordering, he showed how it can be generated by the program with appropriate constraints on the nondeterministic choices being made. Inspired by Edsger's treatment, Jay Misra and I developed a theory of *maximally concurrent programs* that allowed us to reason about concurrent programs that were weakest for a given temporal specification. At its heart, we used the same trick that Edsger had described: derive a constrained program from the original one by adding auxiliary variables.

These are just a couple of examples of the gems that are scattered throughout the EWDs. Toward the end of my graduate life, I was fortunate enough to play a small role in helping set up the Dijkstra archive. To this day, whenever I am in need of inspiration, I go to the archive, and find and read an EWD that I am unfamiliar with. (I love that I can read them in his own handwriting.) They are a constant source of inspiration and delight.

28.7 Don Knuth, Stanford Computer Science Department

I suspect that the first two people in history whose brains were perfectly adapted for computer science were Alan Turing (1912–1954) and Edsger Dijkstra (1930–2002). Thus, it was a great privilege for me to have had many encounters with Edsger, beginning in the early 1960s. My purpose in this note is to share a few of them that still remain fresh in mind as I think about him many decades later.

We first became acquainted through correspondence about programming languages. Jack Merner and I had written a somewhat provocative note called "ALGOL 60 Confidential" [*Communications of the ACM* 4 (1961), 268–272]. Edsger remarked in a letter that he didn't think it was "written under the Christmas tree," and he recommended a less confrontational tone. I tried to keep his advice in mind when I wrote "The remaining trouble spots in ALGOL 60" some years later [*Communications of the ACM* 10 (1967), 611–618].

Both he and I were parttime consultants to Burroughs Corporation during those days, and he crossed the Atlantic at least once to visit their engineers in Pasadena, near my home. One of the many things we discussed at that time was his ingenious idea for a hardware mechanism that would guide an operating system's

Figure 28.3 Edsger Dijkstra and Donald Knuth, photo taken by Jill Knuth at the Knuths' home in Stanford on April 18, 1975.

page replacement algorithm by maintaining a sequence of exponentially decaying bits that recorded recent activity. I don't know if he ever published those thoughts.

My files contain an interesting letter that he wrote on August 29, 1967, enclosing a preliminary copy of EWD209:

> I seem to have launched myself into an effort to develop thinking aids that could increase our programming ability. I expect that in the time to come this effort will occupy my mind completely. (This EWD209 was actually born when I reconsidered the origin of my extreme annoyance with Peter Zilahy Ingerman, who to my taste failed to do justice to Peter Naur in one of his recent reviews. And I started thinking why PZI's attitude seems so damned unhelpful. So his unfair review may have served a purpose, after all!)

In August 1973, I had a chance to visit his home in the Netherlands. By coincidence, two of my most significant mentors, Dick de Bruijn (mathematics) and Edsger Dijkstra (computer science) both lived in Nuenen, a few blocks from each other. Edsger and I spent a delightful hour playing four-hands piano music on his magnificent Bösendorfer piano. (Of course, I let him take the lead in setting a proper tempo, etc.)

A couple days later, I decided to have some fun when I gave a talk at the Technical University of Eindhoven. I discussed what has become known as the "Knuth–Morris–Pratt algorithm," writing it in English step-by-step on the blackboard. When I got to step 4, I paused and pretended to be at a loss; I said, "Hmmm. Is it legal to use the words '**go to**' in this place?" Edsger said, "I saw it coming."

I think he had an encyclopedic knowledge of just about everything. During dinner conversations we rarely discussed computing, and the topics varied widely. I always learned something new from him, whatever the subject.

During January 1993 I stayed a few days with Ria and Edsger at their Austin home. We drove to Pedernales Falls State Park, and I was pleased to see that they both were genuinely in love with Texas.

28.8 Alain J. Martin, California Institute of Technology

I arrived in Eindhoven on a very cold day in January 1971. I had obtained a research fellowship from the French government, and I was on my way to join Dijkstra's research group. I had read his article "Cooperating sequential processes" and was impressed both by the (at the time) new subject matter—concurrency—and by the way in which it was treated.

His research group at the Technical University of Eindhoven was very small, just Coen Bron and Wim Feijen. Martin Rem would join a few months later. It would never be large. The atmosphere was both serious and cozy, studious and informal. I was immediately invited to his home. We soon became friends, and we would always remain close.

As we all know, Edsger's contributions were very varied as he had a talent for identifying important problems and of isolating them in their simplest and most general form. But a topic that might have been the single most important red thread throughout his career was the mathematical development of programs. That aspect of his work had a strong influence on my own research. Through his approach, I saw program notation and the accompanying proof logic become a mathematical method for developing algorithms rather than simply a language for describing an already existing solution.

That was the starting point for my work on VLSI synthesis when I joined Caltech. I was inspired by Edsger's approach to program correctness and systematically derived the final design proceeding through multiple levels of abstraction. Looking back, I am still amazed that something as complex as a microprocessor can be derived from a two-page long program by formal transformations. The abstractions are so well embedded in the final structure of the product that no other way of deriving the design would seem possible. Edsger understood my approach but warned me: "you will never convince them..."

It has been said that geniuses are never one-dimensional in their talents. It was true of him. First, he had a gift for music. He played the piano well, and he had a very fine-tuned ear. But mostly he was extremely gifted for languages. His written Dutch was beautiful, and we have all appreciated his English in his numerous EWDs and personal letters. He also spoke German and more French than he was ready to admit. (He was ostentatiously Gallophobic.) If the Dutch I still speak today has any quality, I certainly owe it to him.

He was always writing, if it wasn't a new EWD, then it was a letter to a friend, or an entry in his journal. His relation to words was very special. He claimed, facetiously, that he was in charge of the "Word Wide Fund" whose mission was to salvage words from extinction . . . In his technical writing, he used language like a precision tool. For him, precision did not necessarily imply ease of reading, and he stated that the reader also had to make an effort to understand. Which once in a while caused friction with some of his co-authors.

About his attachment to the true meaning of words, he once told me the following anecdote. He was taking a driving lesson when the instructor said "rustig, rustig". Edsger did not react. Again: "rustig, rustig". At this point, anybody else would have understood that the instructor meant "slow down!" But in Dutch, "rustig" just means "calm" and Edsger was already perfectly calm...

During Edsger's lifetime, his country, the Netherlands, went through truly enormous social and economic transformations. The world in which the young Dijkstra grew up and the one in which he lived in Eindhoven were a world-war apart and very different. My guess is that, at times, he might have felt a little alien in the modern world. He sometimes gave me the impression that he looked at his fellow human beings as an outsider. Often, on an evening in front of the fireplace with a whisky in hand, he would practice his wit with a perceptive and funny description of human foibles he might have observed recently. He was sharp but not mean.

Edsger and his wife Ria were the perfect example of the saying that behind every great man there is a great woman. In my view, he would not have functioned without her. She was both kind and strong, and provided the warmth and practical sense he was missing. Faced with a difficult decision, it was Ria he consulted with, and he always followed her advice.

He had an aristocratic demeanor. He claimed he believed, quoting E.M. Foster, in an aristocracy of the mind. That was the quality he was looking for in his friends, to whom he was very faithful.

Those are the few memories of Edsger W. Dijkstra I chose to share. I have included some more personal ones, as enough other contributions will cover his scientific achievements, and I had the privilege of knowing him well.

28.9 Jayadev Misra, University of Texas at Austin, Austin, TX

I first met Edsger at a dinner in 1976 when he was visiting Austin as a Burroughs research fellow. I had a lively conversation with him, and by the end of the evening he had agreed to mail me his EWD notes. I met him several times later when he was visiting Austin. I got to know him much better after he joined the Department of Computer Science at the University of Texas at Austin in 1984. I saw him almost every day until he left Austin in 2002, except the summers which he spent in the Netherlands. The last time I saw him was in May 2002 at the airport in Eindhoven; he insisted on seeing me off for a very early-morning flight because "he will never see me again".

I don't remember ever being intimidated by him. I realized that he would have respect for precise arguments and contempt for shoddy ones, and he would not be shy about sharing his views in public. So, I was careful to avoid shoddy arguments in my writing and in speaking to him. He probably saw potential in me and Mani Chandy. He offered to read our papers line by line, with us in attendance, and comment on the writing style, subject matter, and, even, on the worthiness of the paper. Those were exhausting experiences lasting several hours for even a short paper, but rewarding for they often led to new insights and crisper arguments.

Dijkstra railed against anthropomorphism, in particular treating inanimate objects as human beings. He abhorred statements such as "this guy believes that the other guy is sleeping, so he wakes him up and the other guy realizes ..." to describe communications in a computer network. So, Mani and I asked him with great trepidation if he will read our paper "How processes learn" sometime during 1985. He actually loved the paper but asked us to replace "learn" by a neutral word. We ignored his suggestion; marketing considerations prevailed!

About anthropomorphism, here is his sarcastic comment about a defective toilet: "It flushes, but without any enthusiasm". He told me about a ticket agent at an airport who said to him "Mr. Dijkstra, the computer does not know your name". I laughed and quipped "Even if it did, it would have no respect for you". His more serious response was "That was what attracted me to computers in the first place".

Dijkstra introduced the notion of "fairness" in concurrent computing, whereby every participating process takes a computation step eventually, though the speeds of the processes are unspecified. He was very proud of this abstraction whereby he eliminated the relative speeds of the devices in his design decisions in building an operating system. He recalled that his colleagues were horrified that his design did not take into account the differing speeds of the printer and the punching machine; he had responded that the punching machine also makes more noise, and he did not take that aspect into consideration either.

In "The structure of the "THE"-multiprogramming system"[3] he explains the notion of fairness as follows:

> We have given full recognition of the fact that in a single sequential process (such as performed by a sequential automaton) only the time succession of the various states has a logical meaning, but not the actual speed with which the sequential process is performed. Therefore we have arranged the whole system as a society of sequential processes, progressing with undefined speed ratios.

He reemphasized this notion in "Hierarchical ordering of sequential processes"[4], see chapter 23 in this book by Allen Emerson.

Later, though, he became opposed to this notion of fairness. We can only surmise that this change in his viewpoint came about with the publication of "Guarded commands, nondeterminacy, and the formal derivation of programs"[5]. He developed a theory of program correctness in this paper that could handle only bounded nondeterminism, whereas a proper treatment of this form of fairness requires unbounded nondeterminism, see Chapter 8 in this book written by Krzysztof Apt and Ernst-Rüdiger Olderog.

The graduate students in my department used to have a panel discussion every Friday afternoon. They once invited Edsger and me to debate the merits of fairness. Edsger went first, delivering a superb argument in his inimitable style about the unsuitability of waiting for the end of infinity. I had heard many of those arguments before, so I was prepared. I claimed that inspired by Dijkstra's talk, I propose banning irrational numbers and limiting Turing machine tape length to the number of atoms in the universe. I believe I thoroughly demolished his arguments[6]. He shook my hand, formally, after the panel discussion, but did not say anything. He

[3] E. W. Dijkstra. 1968. The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5, 341–346.

[4] E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inform.* 1, 115–138. DOI: https://doi.org/10.1007/BF00289519.

[5] E. W. Dijkstra. 1975. Guarded commands, nondeterminacy and the formal derivation of programs. *Commun. ACM* 18, 8, 453–457. DOI: https://doi.org/10.1145/360933.360975.

[6] K. M. Chandy and J. Misra. July. 1988. Another view on "fairness". *SIGSOFT Softw. Eng. Notes* 13, 3, 20. DOI: https://doi.org/10.1145/51696.51698.

and his wife, Ria, came to our home that evening carrying a bouquet of flowers. Dijkstra simply said, "Marvelous response" and handed me the flowers.

The Year of Programming at Austin during 1986–1987 was one of the highlights of my professional career. It attracted a number of eminent computer scientists, Tony Hoare, Amir Pnueli, and Manfred Broy among them, who spent extended periods in Austin including a year-long sabbatical for Tony. There was a series of conferences attended by many prominent computer scientists from around the world. The director of the very first conference, Tony Hoare, devoted the first day of the conference to Unity, a concurrent programming model that Mani Chandy and I were developing at that time. Unity was then in its nascent stage, so it was a great opportunity to present the ideas at length and defend the work against criticisms thrown from all angles. Edsger contributed mightily to it, teasing, probing, and suggesting. He attended every single talk in every conference in that series.

I had once written a very short proof of Cantor's diagonalization theorem for arbitrary sets using a formal argument, and faxed it to Edsger. After returning from a month-long trip, I received a call from him in which he announced that "You are now a co-author with me". He told me that he has described the steps for a systematic derivation of the proof, and that his submitted handwritten manuscript has been accepted with a few minor revisions[7]. Looking over the paper, I realized how little of my formal proof was based on a flash of intuition and how much on a standard sequence of recognizable steps of which I was completely unaware.

A notable quote (with slight paraphrasing) from Dijkstra's Turing award lecture[8]: "In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind". Though I am hesitant about accepting the first part of the claim—I believe as a tool computers will be as fundamental as electricity—the second claim, about its influence on our culture, can be assessed only at a sufficiently long time in the future. If this prediction comes true, then Edsger himself will have been the inspiration for it. It has been my privilege to learn from a man whose teaching methods follow the methods of Socrates (elenchus) and the Dialogues of his pupil Plato. Their contributions to culture have lasted over two thousand years.

[7] E. W. Dijkstra and J. Misra. 2001. Designing a calculational proof of Cantor's theorem. *Am. Math. Monthly* 108, 5, 440–443. DOI: https://doi.org/10.1080/00029890.2001.11919771.

[8] E. W. Dijkstra. 1972. The humble programmer. *Commun. ACM* 15, 10, 859–866. Turing Award lecture. DOI: https://doi.org/10.1145/355604.361591.

28.10 David A. Naumann, Stevens Institute of Technology, Hoboken, NJ

Around 1994, when I stayed with Ria and Edsger in Nuenen, he took me to a shop in Amsterdam for repair of his fountain pen. That evening he gave me a pen and I said my writing was not worthy. His response, in friendly tone, was: "It may improve." This was neither the first nor the last time I benefited from his faith in people's ability to change.

One of my first personal encounters with Edsger, around 1987, was an opportunity to change my mind: He told me he and Ria decided to move to Texas to show they could learn new tricks. That made me warm to him, overcoming a previous dislike that can be blamed on free pizza when I had just started graduate study at UT. At that time (1984), I was taking courses to compensate for my minimal undergraduate training, and I knew next to nothing about the person who gave a dinner speech at an ACM event. His ranting about the poor-quality work being done on his new home made me dismiss him out of hand. My negative impression was reinforced when I started attending department seminars: UT had recently banned smoking on campus and Edsger was the lone scofflaw among many dozens of students and faculty.

A self-supported Ph.D. student was allowed to be rudderless, and at first I was. Then in Fall 1986 friends steered me to take a course with Tony Hoare, who advised me to take Edsger's course, which changed my life. Through Edsger and Tony, I came to understand programming, which for me had been an enjoyable and lucrative craft, as a scientific discipline. I became enamored of the possibility to derive, by calculation, correct solutions to programming problems.

Edsger showed the efficacy of manipulating symbols by formal rules without thinking much about the interpretation of those symbols. He also made pronouncements about being a formalist, which sounded dogmatic and didn't tell the whole story. During one of the events of the Year of Programming (1987) I had the pleasure of sitting at lunch with Edsger and Bob Boyer who was trying to get Edsger to admit the existence of epsilon-nought, an ordinal number important in constructive mathematics. Bob sketched increasingly complicated "bags" on a napkin. Edsger seemed open-minded, not at all dogmatic, and was friendly to me. It was my first hint that some of Edsger's public behavior was theatrical, drawing attention to himself and his ideas, backed by profound dedication to science and compassion for students.

The following year, in Edsger's course, I complained that his presentation of predicate calculus allowed nonsense like the predicate of Russell's paradox. He said it was an issue he chose not to address. He advocated a strictly formalist position yet seemed to disdain type theories that formalize constructivist

Figure 28.4 Edsger W. Dijkstra and David Naumann. Photo by Ria Dijkstra in Nuenen, The Netherlands, 1994.

mathematics in avoidance of contradiction. Later I came to appreciate what I understand to be an engineer's use of mathematics: We can rely on our interpretations to keep us from nonsense, while getting the most benefit from symbols by neither cluttering the rules nor thinking about interpretation. Edsger's performances in class were stimulating and it was abundantly clear he respected and cared about students. I was shy and insecure, though, and when it came time for the oral exam I did not do well; yet Edsger was patient and warm during the entire meeting. I left feeling encouraged.

By the time Edsger quit smoking I was participating in the Austin Tuesday Afternoon Club (ATAC). The force of his will to change was on display when he quit cold turkey. In the ATAC I often saw that Edsger's strongly held views were open to revision and refinement. He criticized conventional practices in mathematics and logic, but one afternoon after coming to appreciate a particular fine point he said to Allen Emerson, "you logicians are no fools."

Tony had posed the problem for my dissertation: category-theoretic laws of predicate transformers. Edsger, Ralph Back, and Carroll Morgan had made clear the value of predicate transformers as a uniform basis for calculating with specifications and programs. Generalizing their work to higher-order programs seemed fraught with the danger of logical inconsistency, so I spelled out set-theoretic interpretations of everything, in pedantic detail and in contrast to the axiomatic style Edsger was using to develop predicate transformer theory in collaboration with Carel Scholten. Complicated category-theoretic definitions were not to Edsger's taste; he once proposed that I write a paper on why theoretical computer scientists do not need to know category theory. Nonetheless, when it came time to review my draft dissertation (over 200 pages), Edsger read almost every word and formula, as evidenced by extensive margin notes in pencil. He invited me to his home, and we spent many hours going through the text and discussing it in detail. There were only a couple of sharp comments; in connection with some dense and wordy summary paragraphs, he wrote, "This is a style of doing mathematics that I abhor." But I had done a lot of calculations and he wanted to understand everything. Rather than dwell on whether my general approach was a good idea, we discussed the rationale for various design decisions, technical details, and notations. He guided me to many corrections and improvements.

I had entered graduate school in order to pursue my interests in linguistics, with no career plan beyond supporting myself as a programmer. Now approaching graduation (1991), I was offered a position at nearby Southwestern University thanks to Walt Potter who was participating in the Austin Tuesday Afternoon Club. Walt had persuaded his colleagues at Southwestern to teach their introductory programming course following the program derivation approach of Dijkstra and Feijen. Through Edsger's example, I had come to appreciate the importance of the university and the ways academic scientists can contribute to society, so I took up that path myself. As we became friends, Edsger continued to accept me as I was, while helping me improve. Here is one example of his guidance in academic matters: About the course at Southwestern, Edsger said "fools rush in," and advised me to create an introductory course in functional programming to prepare students for imperative program derivation (which I did and it did).

In research I have often made progress on some problem by thinking about how Edsger might have disentangled it and made it amenable to calculational proof. For example, an attempt to use equational reasoning to prove soundness of behavioral subtyping led to the discovery of its completeness.[9] Another example was the derivation of an information flow monitor from its specification.[10] This drew on

[9] G. T. Leavens and D. A. Naumann. 2015. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst.* 37, 4, 13:1–13:88. DOI: https://doi.org/10.1145/2766446.

[10] M. Assaf and D. A. Naumann. 2016. Calculational design of information flow monitors. In *IEEE Computer Security Foundations Symposium.* IEEE, 210–224. DOI: https://doi.org/10.1109/CSF.2016.22.

ideas in a groundbreaking article by Patrick Cousot, which was in turn influenced by Edsger's work.[11]

The last time I saw Edsger was in February 2002, in a hospital in Austin. In our brief conversation he urged me to keep in touch with a researcher whom he and Ria had taken under their wings. Edsger introduced me to the term doctor father and he was one to me.

28.11 J.R. Rao, IBM Thomas J. Watson Research Center, Yorktown Heights, NY

My first awareness of Prof. Edsger W. Dijkstra came when I was an undergraduate freshman at the Indian Institute of Technology Kanpur. His reputation as a deep and influential thinker had preceded him across the globe. As an aspiring computer scientist, I purchased a copy of his classic book titled, *A Discipline of Programming*. As I had been forewarned, I found his writings to be abstruse, requiring a maturity that I lacked. Even so, in a premonitory way, I left the book on my shelf till I came back to it many years later.

In the Fall of 1986, I joined the doctoral program at the University of Texas at Austin. I was excited as the faculty roster comprised several pioneering, world-famous researchers and the department had just launched the Year of Programming. I first ran into Prof. Dijkstra as he was waiting outside Dr. K. Mani Chandy and J. Misra's offices, in his signature Birkenstock sandals, with his long leather purse, smoking a cigarette. I felt intimidated to see him in person and he barely acknowledged my greeting. Sensing his preoccupation and suspecting aloofness, I left.

As part of graduate coursework, I took three consecutive courses taught by Prof. Dijkstra entitled *Capita Selecta—Selected Problems in Computing*. In the very first class, he asked us to write the English alphabet (in both upper and lower cases) and the arithmetic digits and having done so, to reflect if we could distinguish between the lower case "p" and upper case "P", between the upper and lower case "O" and the digit "0", between the lower case "l" and the digit "1" in our handwriting. Thus began a journey in learning how there was no detail that was too minute for our consideration and how no effort would be spared in order to not confuse our readers and ourselves.

Prof. Dijkstra's classes were a joy to attend. For each class, he would choose a programming problem and challenge us to share our solutions on the blackboard.

[11] P. Cousot. 1999. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen (Eds.), *Calculational System Design*. NATO ASI Series F, Vol. 173. IOS Press, Amsterdam, 421–506.

Figure 28.5 Prof. E. W. Dijkstra and J. R. Rao at the Imperial Abbey of Ottobeuren, Bayern, Germany (during an excursion of the Marktoberdorf Summer School in 1988.

He was a kind and gentle teacher who would guide us by suggesting changes to the presented solution. Did we choose our notation correctly for framing the problem and was that particular subscript really necessary? He would repeatedly stress the importance of choosing words carefully: "If you have to use your hands, then there is something wrong with your words", he would say, adding, "imagine there is a blind man in your audience, how would you speak?"

Over time, I came to learn and appreciate that Prof. Dijkstra's class was operating at two different planes. There was the lesson, and then the lesson within the lesson. At one level, the goal was to solve the presented problem. At the second and richer meta-level was the approach for arriving at the solution. Many have commented on his methodical and economic use of the blackboard in his classes. Despite the complexity of problems that were tackled, he was somehow able to fit the solutions in one blackboard. I began to appreciate how careful choice of powerful notation and tools could help us master complexity and communicate our arguments crisply. Prof. Dijkstra taught us that in computing science complexity comes for free; one has to work hard for simplicity. Others commented on how he would not react to questions immediately but wait for a minute or two before responding. I came to appreciate his thoughtfulness as he would examine both the source of the question and confusion before giving a measured reply. In this way, the classroom became a forum for a generation of computing scientists to examine their instinctive approaches to problem-solving and revisit their instinctive methods of reasoning and communication. Our final course grade was based on a one-on-one, face-to-face, final examination with him, conducted using a pen and a few sheets of white paper where he could observe firsthand how our thinking had evolved. Prof. Dijkstra knew very well that to effect foundational change he had to shape the thinking of a new generation of software professionals, and he set about doing so diligently.

I was fortunate enough to be one of the select few students invited to join Prof. Dijkstra's Austin Tuesday Afternoon Club (ATAC). Every Tuesday afternoon, we would read a technical paper examining the technical arguments and comment on how the notations and proofs could be improved. At one of the ATAC sessions, Prof. Dijkstra shared a note from Prof. Robert Tarjan on Vizing's Theorem, who wrote, "I include a proof that is neither clear nor elegant in the hope that you will rise to the challenge and find the right proof". I embraced the opportunity to work on Vizing's Theorem and was able to propose a simpler solution in the next ATAC meeting. Much to my delight, Prof. Dijkstra invited me to collaborate on a manuscript describing the solution. Not only did this give me an opportunity to work together with him but it also gave me insights into the methodical and disciplined manner in which the master approached his work.

So, on February 21, 1990, we began work early, at his study desk in his home. We worked steadily together, with pen and paper, visiting each and every detail of the argument to ensure that the simplest and most elegant proof was crafted. Prof. Dijkstra heard me with his characteristic patience, adopting some of my ideas and suggestions, while explaining why he discarded others. The argument emerged iteratively; several preliminary drafts were discarded. After a light lunch and a walk through their neighborhood together, we resumed working, and by dinner time a first draft of the manuscript, EWD1075, was ready. This was revised and refined further twice, mainly by Prof. Dijkstra as EWD1082 and EWD1082a. It gave me a great sense of professional satisfaction to witness firsthand how one of the great minds in our field worked and I imbibed some of those practices in my approach to my work as well.

After graduation, I joined IBM Research in 1992. Compared with the methodical thinking and interactions of my academic upbringing, the culture of the industrial research environment was very different: it was replete with fast-talking colleagues whose mannerisms were influenced by their work and the complexities of the systems that they were building.

The organization found value in many of the skills that I had acquired as a graduate student, though not in the ways that I expected. While my work did not have any impact on the way software is developed at IBM, practitioners welcomed some of my simple solutions to programming problems they encountered. My presentations, designed to be crisp and simple, were appreciated by senior management as it did not overwhelm them with a morass of detail. Ironically, this made me a leader in their eyes and despite my reluctance I was promoted to technical leadership and management positions. I truly feel that, in many ways, the success that I have enjoyed at IBM Research stemmed from the teachings and values of Prof. Dijkstra, Prof. Misra, Prof. Apt, and others at the University of Texas.

The problems that Prof. Dijkstra wrote eloquently about continue to persist today. While our academic institutions and software industry have been able to work around these issues with no significant penalty to the scale of software systems that are built, the problems still remain. Three decades later, when I see how Programming is taught to introductory Computer Science students at elite institutions of learning and I see textbooks on Programming, I know that we are raising a generation who may be adept at today's tools and are quick to program a solution—but lacking a re-examination of their thought processes may not be able to improve their capabilities significantly. Given the scale of human ambition, which ever aspires and spirals higher, it is just a matter of time before we will need even more advanced techniques. For instance, how are we going to automate reasoning to support the resurgence of Artificial Intelligence when we are still in the process of learning how humans reason? After three decades in industrial research, I am convinced that good work never goes to waste and that it is only a matter of time before we will come back to the foundational work that Prof. Dijkstra began.

I kept in touch with Prof. Dijkstra and shared my professional experiences with him. We would communicate by physical mail since he didn't use e-mail. He was a faithful correspondent whose beautiful, multipage handwritten responses I still have. His letters were a delightful mix of professional observations, technical issues, life lessons, and witty, personal observations. One of them was completely written using his left hand. He often urged me to continue the explorations that we had begun together. All of them reflected a deep care and compassion that I had wrongly mistaken for an aloofness in our first encounter.

My last discussion with him was in August 2002, just a few days before he passed away. Sensing that I was distraught, he said, "When you hear of my passing, you will be sad and tears will come to your eyes but then you should let the moment pass and continue with your life for my life is one that should be celebrated and not be mourned."

I will always remember Prof. Edsger W. Dijkstra as an inspirational teacher and pioneering researcher who had an immeasurably profound influence on my life.

28.12 Hamilton Richards

In 1976, a freshly minted Ph.D. looking for a job, I received an offer from Burroughs Corporation. Its B5000 series had earned the approval of the acclaimed Turing Awardee Edsger W. Dijkstra, who was the company's Research Fellow. This convinced me to join the Burroughs laboratory in San Diego that was investigating dataflow computing.

Edsger's fellowship entailed consulting visits to Burroughs plants and labs. During his first visit to the San Diego lab after my arrival, Edsger noticed a binder in my bookshelf, labeled "E.W. Dijkstra," containing photocopies I had received as a graduate student. He remarked, "This is how authors are cheated out of their royalties." It was a rough start for a friendship whose significance in my life was exceeded only by my marriage to Joanne.

In 1978, the dataflow project was canceled, and I moved to Austin as a founding member of the Burroughs Austin Research Center (BARC) investigating functional programming (Robert Barton, the B5000's chief architect, had been inspired by the Church–Rosser theorem's implications for concurrent processing).

During Edsger's first Austin visit he complained about his hotel room's lousy writing table, so I invited him to do his writing at our house. In subsequent visits, he accepted our invitation to stay at "Hotel Richards". Conversations at dinner and into the evening were engrossing, ranging widely over politics, culture, academia, technology, and the English language. He became a fan of *The New Yorker*—especially the cartoons. We fondly remember Edsger stretched out on the living room carpet, reading contentedly while our 5-year-old son covered him with sofa cushions.

At BARC we had lively discussions with Edsger about proving correctness of purely functional programs (e.g., EWD825, EWD827). At first Edsger contended that although functional programs are typically much shorter than their imperative counterparts, their correctness proofs would be longer. Further investigation refuted this supposition, and in later years Edsger's appreciation of functional programming was revealed in a memo [0] defending the use of the functional language Haskell in my introductory programming course.

One of Edsger's visits coincided with a visit by David Turner, who was consulting for BARC. Edsger was in the audience for one of David's lectures, and in the style for which he was known, he began to interrupt with questions and objections. Given the two towering personalities, escalation was inevitable, and finally David aborted his lecture. At dinner that night, Edsger conceded that he'd gone a bit over the line, and in subsequent days at the lab he and David got along well.

BARC had a no-smoking policy, but recognizing Edsger's smoking habit we set up an office for him with special ventilation. His response was a blistering memo attacking society's growing disapproval of smoking. A few years later he quit smoking, so inconspicuously that three days passed before his wife, Ria, noticed.

On every visit to Austin, Edsger was invited to give lectures at the University of Texas. As Burroughs support for his work declined, the visits began to include interviews, and in 1984 he was appointed to the Schlumberger Centennial Chair in Computing Science.

That meant moving to Austin and finding a house. Edsger's stays at Hotel Richards having familiarized him with the neighborhood, he and Ria chose a house just up the hill from ours. It was a new house, requiring modifications including one to make space for the Bösendorfer grand piano. Staying with us for six weeks, they became de facto members of our family.

Edsger's love for Texas—the wide blue skies, the friendly Texans, and the landscapes' variety—is captured in the T-shirt he often wore bearing the motto, "Life's Too Short Not To Live it as a Texan." He and Ria toured widely in their VW camper, dubbed "Harvey the RV" or "the Touring Machine." Staying in state parks, Ria would cook while Edsger would write.

Their regular after-dinner walks in the neighborhood frequently passed by our house, and they would stop in several times a week for conversation, iced coffee, and—for Edsger—a glass of scotch whisky.

In 1986 BARC closed, and Edsger recommended me to coordinate UT's Year of Programming, a US Navy-sponsored series of workshops. That was a half-time position, so I was assigned to teach for the other half. Thus began my 18-year career as a Senior Lecturer. The subjects I covered in my courses included Edsger's calculational reasoning and weakest-precondition methodologies, but his most important influence on me is summed up in his epigram:

> ...if 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself, Dijkstra would not have liked this, well that would be enough immortality for me.

After years of digestive problems, in early 2002 Edsger was found to have esophageal cancer. Outpatient treatment failed spectacularly—in his first day home from the hospital, wearing a chemotherapy pump, he collapsed so violently that his head made a huge dent in the bathroom wall. After weeks in hospital, he was pronounced (barely) fit to travel. Home for a day, he and Ria hosted a crowd of visitors at lunch with barbecue from the nearby County Line, and he played a little Mozart on the Bösendorfer. Then they flew home to Nuenen, accompanied by me as logistical assistant.

Returning to Austin, I undertook the task, with help from my wife and Jay Misra, of closing up the Dijkstras' Austin household. What was not sold was shipped to Nuenen (including the Bösendorfer). The job was done in time for us to leave for our summer vacation in New Hampshire. I neglected to take my passport, so that when the word came that Edsger had died, there was no way for me to get to Nuenen in time for the memorial service. The regret lingers on.

28.13 Mark Scheevel, Vericast, Austin, TX

After receiving degrees from Rice University in 1978 and 1979, I moved to Austin where Burroughs Corporation had established a research center to study approaches to parallel processing, among other things. We were studying functional programming languages as a means to achieve significant parallelism without requiring explicit synchronization and coordination by a programmer.

Although it may not be widely remembered today, Edsger was a Burroughs Corporation research fellow from 1973 to 1984. He also spent quite a bit of time in Austin, eventually accepting a position at the University of Texas in 1984. This meant that he was a frequent visitor to our research center, and he spent many hours educating us and critiquing our work.

As was common at the time, I came to computer science via electrical engineering (in fact, Rice didn't formally establish a department of computer science

[0]. To the members of the Budget Council, Edsger W. Dijkstra, 12 April 2001. https://www.cs.utexas.edu/users/EWD/OtherDocs/To%20the%20Budget%20Council%20concerning%20Haskell.pdf.

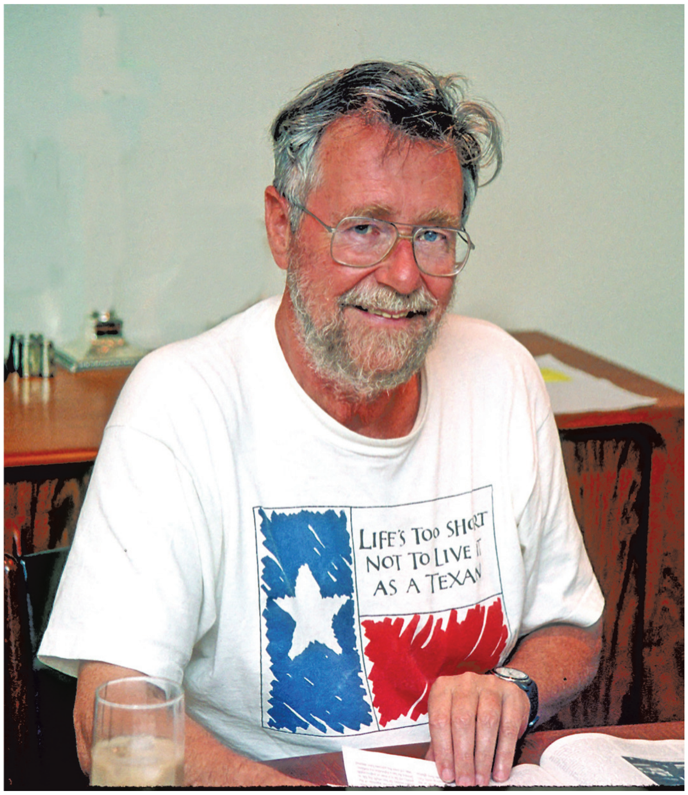


Figure 28.6 Edsger W. Dijkstra enjoying after-dinner conversation in March 1995 at the Austin home of Hamilton Richards, wearing a shirt that testifies to his contentment in Texas.

until 1984). That meant that I did a substantial amount of assembly language programming in addition to programming in higher-level languages, and I developed a tendency to regard programs as something one hammered into shape over time. Edsger introduced me to the idea that programs could be treated as mathematical entities, with properties that could be subjected to mathematical analyses, and that there were ways to prove assertions about the properties of a program.

Those are powerful ideas, and they had a profound effect on the way I approached problems. In my career, I have worked on compilers, interpreters, query languages, and query optimizers, and in each of these areas these ideas have been invaluable. At an abstract level, each of these is about translating a source language to a target language while maintaining the original semantics. The lessons that Edsger taught us about systematic program transformation provided me with a solid foundation for this work. And his insistence on truly understanding the problem one is trying to solve and always searching to generalize solutions caused me to raise my personal standards.

I have had many influences in my career—and I am indebted to Burroughs colleagues who helped a young engineer find his footing—but I was truly fortunate to have the opportunity to interact with Edsger during my early, formative years. He made me a better engineer.

28.14 David Turner, University of Kent, UK

My personal encounters with Edsger Dijkstra date from 1980 and began by letter. I received a note from Edsger dated May 7, 1980, enclosing a copy of EWD735, "A mild variant of combinatory logic", which he sent to me at the suggestion of Ham Richards, relating to my January 1979 article in *Software: Practice and Experience*. Edsger had been sufficiently interested in the topic to set about inventing his own version of combinatory logic as a way of better understanding it. In the note he also enquired if I would be at the Burroughs facility in Austin, TX, in August when he expected to be there.

I was to meet Edsger much sooner. I had been invited to give seminars at six Dutch computer science departments on a visit organized for me by Doaitse Swierstra at Groningen, and on Tuesday, May 20, I found myself addressing Edsger's Tuesday afternoon club at Eindhoven. After the discussions had finished, Edsger insisted on my staying over with himself and Ria at their house in Nuenen. They were gracious hosts there, as I found them to be later in Austin, and 40 years on I recall the evening with pleasure. They had two very large dogs, creatures of which I am normally wary, but the Afghan hounds fortunately proved docile. Edsger enquired about my family background and appeared satisfied to discover that my father was a businessman. Edsger was interested in people. His trip reports frequently contain acute sociological observations, and while famously intolerant of what he judged bad science or sloppy reasoning, Edsger was in my observation never an intellectual or social snob.

I might have anticipated the dogs, as they appear in *A Discipline of Programming* (1976) p. 65, where Edsger writes "When the youngest of our two dogs was only a few months old I walked with both of them one evening".

An idiosyncrasy that I noticed in the morning at breakfast was that Edsger wore two watches, one on each wrist. One, I think the left, was set to the local time of whatever time zone he was in, the other was always on Dutch time. Edsger referred to this, humorously, as "God's time", although from the indications available to me and reported by others I am fairly sure Dijkstra was a nonbeliever. He was, however, very definitely Dutch.

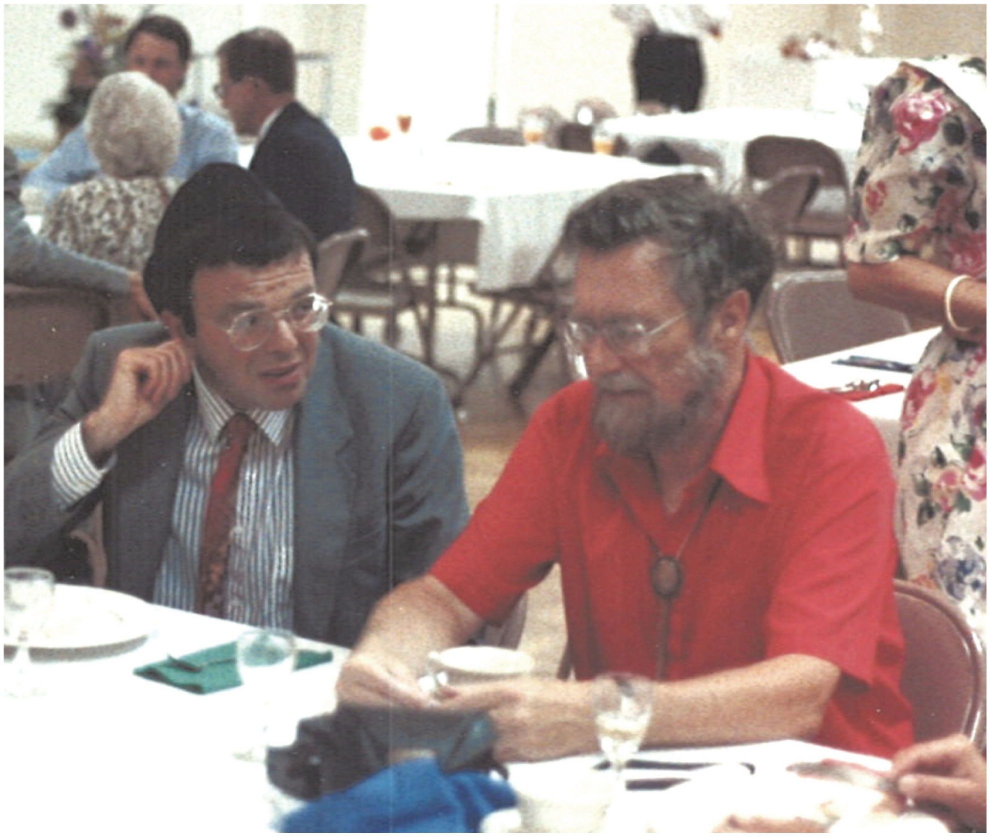


Figure 28.7 Edsger Dijkstra and David Turner during the Bar Mitzvah of Ham Richards' son. Photo taken by Ham Richards in Austin in 1990.

I met Edsger again that year, in August, at Burroughs Austin Research Center, as his letter had anticipated. He took an immediate interest in SASL, a simple lazy functional language that Burroughs had adopted for the project at BARC, and set a problem which we solved together, to generate the decimal digits of "e" as an infinite list. Edsger contributed a crucial lemma, without which my program could not have worked. He wasn't comfortable with "infinite list" by the way, preferring "potentially infinite" or something similar. I wondered if he had been influenced by the intuitionistic school of mathematics, whose members reject the idea of already completed infinities. Years later, I had the opportunity to ask Dijkstra if he had known Brouwer—apparently, they did overlap, both being members of the Dutch Academy of Sciences. Edsger said that Brouwer was the most argumentative person he had ever known (these may not have been the exact words but that was the sentiment), which coming from Edsger was a strong statement.

Figure 28.8 Manfred Broy, Edsger Dijkstra, and David Gries during a skit in the Marktoberdorf Summer School 1998.

I was honored with a second letter from Dijkstra, in November 1980—EWD759, "A somewhat open letter to D.A. Turner", asking for a proof of a certain property of a recursive function. My response is reproduced, with Edsger's comments, in EWD770.

I had the privilege of interacting with Dijkstra quite often in the period from 1980 to 1984 when he ceased to be Burroughs Research Fellow and took up a Chair at the University of Texas. I was a consultant to the functional programming and combinator reduction machine project at Burroughs Austin Research Center from January 1980 until it was, sadly, closed in 1986 following the merger of Burroughs with Sperry. Edsger took a definite interest in this project and sometimes became quite involved during his visits. I recall that Mark Scheevel devised a better method of extracting combinators from SASL "where" expressions, which made it practical to use a copying version of the "Y" combinator in place of a cyclic one, and thus a reference count garbage collector. Edsger became really excited at this development. He was certainly interested in the practical problems of computer engineering, as is apparent from chapters in his career.

I was able to interact with Dijkstra from time to time in the years that followed but regrettably much less often than in the early '80s. I recall another occasion when I dined with Edsger and Ria, this time at their house in Austin. I was a visiting professor at UT for the Spring semester of 1992, to which I had come without my family. My wife had given me a rather nice, remarkably small, portable CD player that I thought would interest Edsger, which it did. He insisted that I borrow whatever I liked from his extensive classical music collection.

Edsger was a truly unusual person. He had many idiosyncrasies: for example, his insistence on numbering from zero; his dislike of canned music—which he would sometimes take direct action to eliminate at source with varying results as I witnessed on several occasions; making his own ink. But for me his most striking property was absolute intellectual honesty. I have an invisible Edsger inside my head that looks over my shoulder when I am writing and quietly goes "Tut tut" if I write something that is muddled or not accurate. I don't always listen to that voice but know I should. Some thought Edsger arrogant. He was dismissive of work he thought unworthy of attention, a category that for him was quite large. But to achieve as much as he did probably requires the ability to cut out noise. For someone of such high intellectual gifts, Dijkstra was remarkably modest.

PART V

VARIA

29

Edsger W. Dijkstra as a Lecturer in Marktoberdorf

Manfred Broy

Edsger Wybe Dijkstra was a key lecturer and a director of the Marktoberdorf Summer Schools from 1970 till 2000. He deeply influenced the spirit of these schools and their scientific topics. Over the years, a vivid interaction took place between his lectures and his scientific interests, the course of which was in turn influenced by the discussions that took place in Marktoberdorf.

29.1 The Idea Leading to the Marktoberdorf Summer School

The basic idea of the Marktoberdorf Summer Schools was quite simple and straightforward. The goal was to invite and bring together a number of young scientists to a two-week international summer school, in particular postdocs and Ph.D. students from the field of computer science, and a group of lecturers who had interesting ideas about scientific foundations of methods for software development.

The Marktoberdorf Summer School was one of the reactions to the discussions and conclusions of the NATO Software Engineering Conferences held in 1968 in Garmisch-Partenkirchen (see Naur and Randell [1969]) and in 1969 in Rome. One of their conclusions was that the development of software is typically an engineering discipline, where engineering is characterized by the fact that it is based on scientific methods. Another conclusion was that there were not enough scientific foundations so far (at the time of 1968), and that building software was based to a large extent on “trial and error” which led directly, because software was becoming larger and more important, to what was called the software crisis. One idea was to build a software engineering institute in Europe, but it turned out to be unsuccessful. As a result, a number of key figures of these software engineering conferences, among them Fritz Bauer, thought about specific ways to come up with scientific foundations. So the idea was born to run summer schools on software engineering and its foundations. In the following, we deal with the role of Edsger as a lecturer and director of the Marktoberdorf Summer Schools.

The idea behind these Summer Schools was that lecturers presented there their recent scientific ideas and results and that there was enough room for discussions of these ideas with the participants and also between the lecturers. The plan was to have highly international events gathering people from all over the world, as lecturers and participants, who could contribute to the formation of an international discipline of software engineering—both concerning education and research.

29.2 "State of the Art" at the Beginning of the '70s

Certainly, it was not the case that there were no foundations of programming and software development at the beginning of the '70s. After the early start of program-based computing in the '40s and '50s, when researchers were mainly concerned about computing machineries and not so much about programming and programming languages, in the second half of the '50s and also in the '60s there were closely some activities to provide foundations for this young discipline. These foundations were related to programming languages: Algol 60 was a first step into a programming language created by an international committee, and it turned out that the work on the language was also influencing specific language features, such as the block concept, procedures including recursive ones, type systems, data structures, and control structures. Actually, Algol 60 was not the first language of the type, it had predecessors such as Fortran or COBOL, and also Algol 58. The work on programming languages triggered in turn debates and research on programming methodology since on one hand how the different language constructs were to be used in the development of programs was a relevant topic and, on the other hand, the development of compilers for programming languages proved to be a challenging programming task.

Working in international committees on these programming languages, intricate questions had to be considered, for instance, understanding of the difference between the "call by value" and "call by name" parameter mechanisms, the role of side effects and the scope concept, and much more. Although a lot of concepts had counterparts in mathematical logics, for many of them the correspondence was not so apparent from the early results in the field. Some of the ideas in programming languages were actually quite new.

In the '60s, there was quite some scientific research triggered by the design of programming languages. In particular, by investigating the meaning of recursion in programming languages, the idea of providing precise, even formal, semantics took shape. At the same time, it was discovered more and more that some of the early results in computing theory—some of which were even discovered before the first program stored machines were built—were relevant for the foundation of programming. Thus, it was already visible (and even more visible today) that there is a kernel in the theory that computer science includes concepts such as computability, computational complexity, and the expressive power of various programming constructs, including quite intricate constructions such as the **go to** statements. These attempts were more theoretical in nature and perhaps not so much motivated by programming methodology and software engineering.

Nevertheless, there were also some work on methodology. There was the early work of Robert Floyd [1967] and Tony Hoare [1969] on assertion logics, while the work of John McCarthy [1963] aimed at understanding the concepts of computability and computation. McCarthy also created LISP with its close relation to the λ-calculus. This led in turn to a number of more theoretical questions such as whether there is a model of the untyped λ-calculus, settled by Dana Scott [1972], and a closely related problem on the meaning of recursion, for which the idea of fixpoints was used, notably by Jaco de Bakker [1971], Dana Scott [1972], and Hans Bekic [1984]. Subsequently Zohar Manna [1974] showed in more detail how to establish properties of recursively defined functions and procedures. This became a basis for some work on transformational programming as advocated by Friedrich L. Bauer, Rod Burstall, and John Darlington (see Bauer [1976] and Burstall and Darlington [1977]).

Following his pioneering work on a compiler for Algol 60 and his subsequent work on design and implementation of the THE operating system, Edsger became more and more interested in understanding how to obtain a clearer program structure when looking at concepts in operating systems that were related and connected to concurrency. He developed the idea of semaphores and described them using a kind of semiformal semantics. It became an early concept of system programming and a first step towards the foundation of the field of operating systems.

It was around this time that Edsger became involved with the Marktoberdorf Summer Schools; he was among the lecturers of the 2nd edition of the school. At that time, he was already a kind of a guru in his field, deeply involved in a number of “battles” with people working in large companies, such as IBM, with their own specific way of programming, a task he considered as a discipline of mathematical engineering.

An important part of his work and interest was devoted to solving algorithmic problems. He liked very much to study usually small, but intricate, problems that were difficult to formalize and difficult to solve. This became to a large extent the contents of his teaching at Marktoberdorf. He came to the Summer Schools with a number of specific “puzzles,” often closely related to programming but also concerned with a correct understanding of the structure of a problem, where the challenge was to obtain a good way of presenting it, which in turn could give the right angle towards finding a solution.

And this fascinated us young scientists because he worked not just by looking at the problems but also by analyzing the ways other researchers dealt with these problems. From the very beginning, Edsger did not very much separate a problem from the people dealing with it. Many of his observations and remarks were about the way people attacked a problem and about their way of thinking. This was always an interesting aspect, and very different from the way other lecturers were presenting their work. Edsger did not hesitate to criticize the lectures of other lecturers, often in quite strong terms. As a result, he was often viewed as an unforgiving person, highly critical, occasionally very personal, and sometimes overly belligerent beyond the standards of politeness.

29.3 Marktoberdorf Summer Schools: The Beginnings

The first summer school was held in 1970. The lecturers of this school were the professors Ole-Johan Dahl, Gerhard Goos, Tony Hoare, David Jerome Kuck, John G. Laski as a guest speaker, and Brian Randell. Dahl was teaching on the block concept, and Goos on communication in process structures, a topic based on the work of Edsger on cooperating sequential processes. Hoare was teaching on abstract data structures and their representations and also on program proving, while Kuck lectured on memories of fast machines as well as on memory hierarchies. Finally, Laski was teaching on code generation, and Randell on processes and data structure. Much of their work was also heavily influenced by the work of Edsger.

At that time, Edsger was already a rising star in the community of the scientific foundations of software engineering. He developed an original style of describing challenging problems in this field by illustrative examples such as the dining philosophers problem.

During the second Marktoberdorf Summer School, held in 1971, Edsger lectured on the hierarchical order of sequential processes. In his course description he wrote (see EWD310 that appeared as Dijkstra [1971]):

> In the very old days, machines were strictly sequential, they were controlled by what was called "a program" but could be called very adequately "a sequential program". Characteristic for such machines is that when the same program is executed twice—with the same input data, if any—both times the same sequence of actions will be evoked. In particular: transport of information to or from peripherals was performed as a program-controlled activity of the central processor.

In his lectures, Edsger brought up an important feature of synchronization and concurrency: they are an inherent source of nondeterminism. Indeed, outcomes of some programs can crucially depend on timing and coordination, which are not completely determined by the program code. Nondeterminism led him into a new direction: nondeterministic programs.

Edsger was already quite well known and had a good sense of how to reach the hearts of the scientific and industrial audience. He became increasingly more famous for his carefully stated but often radical positions characterized by his very strict and careful thinking. He was convinced that a lot of work published in the field at that time (and also later) was not worked out well enough and not thought through sufficiently well, and that a lot of "half-baked" concepts were around.

29.4 Edsger as a Director of the Marktoberdorf Summer Schools

For nearly two decades Edsger served as one of the directors of the Marktoberdorf summer schools. Those who did not know him would not have expected how important this role was for him and how carefully and seriously he took it.

One important task for the directors of the summer school was to define the topics and to select the lecturers. This was done in discussions between us, meaning often Fritz Bauer, Edsger, and myself. We also invited Tony Hoare and a few others to come up with interesting topics and a list of potential speakers.

The second task was to select participants for the summer schools. Usually, we had up to 200 applications, and we were selecting about 80 participants out of them. This task was also taken very seriously by Edsger. Typically, he would come to Munich and we would sit together for a whole day going through the applications, classifying the participants, accepting a number of them in the first run, putting aside some of them as not acceptable, and then finally carefully going through the remaining set to take the final decision. This careful selection of participants made sure that we always had an audience of very high scientific quality, which was certainly a challenge for the lecturers. To quote Edsger:

> The day was spent at the selection of the participants for the next Marktoberdorf Summer School. We had to select 90 participants from 160 applications. (EWD885 [Dijkstra 1984a])

As a director, Edsger carefully watched the schedule of the lectures, helped to make sure that lecturers ended in time, explained to the participants the role of the discussions, and also intervened by asking probing questions so that the lecturers improved presentation of their material. I was impressed how seriously he was taking all this and how much effort he was putting in improving our school so that it could best serve all involved, the lecturers and the participants. This became an important and a significant contribution to the atmosphere of the summer school.

29.5 The Arena of the Marktoberdorf Summer School

Marktoberdorf summer schools were pretty different from conferences and workshops in the field. First of all, among the about 80 participants representatives from all over the world, we had typically from 20 to 30 nationalities. Secondly, thanks to a careful selection, the bright young scientists were the "crème" of the Ph.D.s and postdocs from computer science.

The lecturers were also carefully selected. From the beginning we succeeded in inviting the most prominent computer scientists on a worldwide level; several of them were Turing Award winners. Typically, we invited scientists who were working on new exciting subjects or had just published a book, and as a result could present new approaches in the field.

Most of the lectures were attended not only by the student participants but also by the other lecturers. In fact, the summer school only to a certain extent consisted of lectures for the participants; it was also an "arena" in which intensive discussions between the lecturers took place.

The lecturers had several hours during which they could present their subject. This was substantially more time than during other, "typical" conferences or workshops. Consequently, the lecturers could go into sufficient detail in their presentations. At the same time, other lecturers were listening carefully, often in a critical way. Sometimes different approaches seemed to be in competition, in spite of the fact that science should not be like sport and should not be understood as a form of competition. Nevertheless, this is how it turned out to some extent.

One of the persons who, obviously, liked the disputes and careful arguments was Edsger. He did not hesitate to interrupt lecturers and to ask questions, not only about technical points. In the early days of Marktoberdorf giving lectures meant presenting a lot of handwritten slides. So some of Edsger's questions referred to the handwriting, some to the punctuation, and some others to the way people spoke or wrote formulas. Occasionally, his questions seemed small-minded to the extent some participants sometimes wondered what it was all about. In some situations, one could get an impression that he tried to destroy the presentation of the lecturers. On the other hand, it became quite clear that he was as careful about other presentations as he was about his own lectures. As a result, some of the debates with other lecturers developed into a kind of confrontation and one could see that it was not only a confrontation about scientific matters; it was much more. It was a confrontation concerned with personal views and basic philosophy.

At the beginning of his lectures or at the beginning of a discussion, Edsger used to write short sentences or quotes on a blackboard. Sometimes these sentences were a bit difficult to relate to what was going on. In addition, during his lectures and also during discussions, he would sometimes tell little stories about how he usually carried out his work. Of course, this was completely in contrast to other lectures. Also, most of the time he did not prepare his slides in advance; in the ’70s and ’80s, he wrote his slides during his lectures. When he was lecturing about the development of programs, he was perfectly prepared to present the algorithms this way, sometimes dealing with pretty complex and intricate problems. He carefully spelled out the arguments that led to the conclusion and also established the correctness of what he was writing down. It was extremely beautiful.

One of the stories he told us was about Rainer Maria Rilke, who had—as Edsger explained to the audience—the habit of writing letters by hand, but in case he misspelled something he would throw away the whole writing and started from scratch. Edsger told us that after he had told this story to his wife, Ria, she suggested that he should try to do the same. And so Edsger had a hard time for a few days, each time throwing away a piece of paper when he miswrote something. Since then, he could concentrate hard enough to be able to write letters and texts from scratch, without any mistakes. And, as he told us, that was the way he had been writing since then—not only letters but also scientific papers. He would think a long time about the subject and then he could write down a paper in one go, from the beginning till the end.

Participants were quite impressed by this story. However, as it turned out, Edsger did not always live up to his own standards. I remember well one of his lectures given a day after a long evening with a good Bavarian beer. Usually, Edsger gave the first lecture of the day, and when he began his lecture seemed to be a bit under the influence of the last evening. Since he had told us that he would never miswrite anything, I decided to test this by exchanging the water-soluble pens for the overhead projector by the water-resistant ones. (As a young scientist I was responsible for preparing the lecture material.) Edsger realized after some time during his lecture that he made a mistake in his writing and tried to erase it by making his fingers wet, but this did not work. I was sitting in the first row and asked him whether he should not throw away the whole slide as he told us the other day, and start from scratch. Perhaps this was a bit of a rough joke on my side, but Edsger understood it and accepted it, and laughed about it.

29.6 The Challenge

As a young scientist, and later as a lecturer in Marktoberdorf, I always found it a profound challenge to discuss and to lecture in front of Edsger. Usually, he was sitting in the first row, listening carefully, asking questions when he had the feeling that something was not properly explained, and—in many cases—also expressing his criticism, disapproval, and dissatisfaction of the subject or of the way it was presented. This made lecturing a real challenge. On the other hand, Edsger could also be very kind. After their lectures, he would sit for a long time with the lecturers to discuss how their presentation could be improved. He often seemed to be more interested in the presentation itself than in the presented subject.

One of the highlights of the summer schools were the discussion periods. Every day there were six or seven lectures and at the end of the day there was a one-hour discussion period. All lecturers of the day had to show up on stage so that participants could ask questions. As soon as they were answered, the lecturers were allowed to ask questions themselves or to give comments about their own lectures or about those of their colleagues.

These discussions of the day were often more enlightening than the lectures themselves, in particular because during them one could see the differences between the attitudes of the lecturers and their goals. Typically, many lecturers did not care so much about the lectures of other colleagues; they were mainly concentrated on their own presentations and on how their material could be explained and defended. Not so Edsger. He often seemed to be interested not only in the lectures of the others but also in the attitudes of the lecturers. He tried to understand but also to challenge and criticize their motivation, and to relate the subjects of their lectures to their personal background.

One day he claimed that just by listening to a lecture he could determine the religious background of the speaker, for instance whether he or she was a Catholic, a Protestant, or a Jew. He emphasized that in his view this particular background in education was deeply influencing the style of doing research and lecturing.

I was a bit skeptical as I remembered a story by Kurt Tucholsky with the title “In der Hotelhalle.” The narrator happens to meet a psychologist who claims that—just by looking at the people sitting in the lobby of the hotel—he could identify their personalities and their background. The psychologist tells the narrator his detailed assessments of the personalities of several hotel guests, but after the psychologist says goodbye, the narrator goes to the hotel porter and asks him about the background of these guests. It turns out that all the speculations of the psychologist were wrong. So when Edsger made the above claim, I could not resist and sent him the text of Kurt Tucholsky, but I never got any comment from him on this.

Still, Edsger succeeded in turning the summer school into an interesting challenge for engaged young scientists. He would often introduce some “puzzles” and little problems and then, of course, the rule of the game was to be very fast and quick in understanding the problem and in coming up with a solution. Obviously, Edsger played a game he himself liked a lot. One of his deep pleasures was to look at a complicated and intricate problem, which was in some sense limited, though quite sophisticated—and then to think about it long enough to come up with some interesting observation or an idea leading towards a solution.

29.7 The Role of Formal Mathematics in Programming

One of the interesting and fascinating observations about Edsger is how in the period starting in the '60s and ending at the beginning of this century he changed both his mode of working, in particular as a lecturer in Marktoberdorf, and his views of computer science. With his background as a mathematician and physicist, he was confronted in the middle of the '50s with the course on programming he took—and he was fascinated by that. Typical for him and for other computer scientists at that time was a struggle with the choice of notation for programs, with the satisfactory way of writing programs, and the way one should think about programming concepts and notations well suited both for programming and for executing programs on a computer.

At the beginning, the relationship between the programming constructs and the execution of programs on computers was of paramount importance. As already mentioned, Edsger was involved in the sixties in a number of practical projects such as the Algol 60 compiler and the THE operating system. At the same time, he was teaching courses on programming and started to think deeply about programming concepts and programming methodology. This included the structuring of software, one of the topics he was most interested in at the time he took part in the Garmisch-Partenkirchen conference on software engineering in 1968. His careful thoughts about programming made him one of the key figures, perhaps the key figure, involved in structured programming. In addition, at the end of the '60s and the beginning of the '70s, his sharp thinking, his very particular way of lecturing, and his highly critical views about what was called the practical way of doing software programming made him well known, even famous.

His Turing Award lecture in 1972, titled *The Humble Programmer*, included a large number of observations on the development of programs. His lecture made it clear how deeply he was committed to the problem of choice of the right programming constructs and to the question of whether the style of writing programs relates to the question of whether they are intellectually manageable.

At the same time, in the '60s and the '70s, various attempts were made by computer scientists to gain a deeper understanding of the meaning of programming languages by developing various semantics and proof theoretical approaches, and also to relate them to the task of programming. Perhaps the most important contributions were made by researchers such as Robert Floyd and Tony Hoare in their already mentioned works on formal proofs of program correctness based on the use of assertions. The most important outcome of this work was that it became clear that there was a close connection between logical formulas, such as assertions, and the statements of programming languages.

Being aware of these works, Edsger became more and more interested in logic. Having seen Hoare's approach (which he did not like too much in the beginning), he soon recognized that this was a way to close the gap between informal arguments about program correctness and a proper formal mathematics style calculus for reasoning about programs. As a reaction, he began his own formal approach that turned the assertion-based logic of Tony Hoare into the weakest precondition calculus for his guarded commands language. Guarded commands included nondeterminism and were defined by Edsger's postulational or axiomatic semantics in terms of predicate transformers. Nondeterminism supports the formulation of sets or schemes of deterministic programs and was certainly influenced by Edsger's former work on parallel programs. They were motivated as a step towards developing specifications or program schemes instead of deterministic programs. They supported a more effective use of refinement. Predicate transformers are more calculational (compute weakest preconditions given statements and postconditions) and thus provide calculational support for program development. A further strong advantage was the distinction between weakest liberal preconditions and weakest precondition that supported the additional verification of termination. Edsger applied this approach in his wonderful book *A Discipline of Programming*. There he showed that using a small logical calculus of predicate transformers one could develop small and medium size programs with a rigor and beauty that was beyond what has been shown before.

The first time I met Edsger was two years after he had published his book. In 1978 Edsger gave wonderful lectures at Marktoberdorf during which he explained how to formally develop little programs. He showed all the necessary reasoning needed to develop programs and their correctness proofs, hand-in-hand, starting from the specifications.

Being an independent thinker, he soon developed a view that the way logic was done by the famous logicians was not the right one and not helpful enough for programmers. Consequently, he decided to substantially revise the foundations and structure of logics he wanted to rely on. This eventually led to the book *Predicate Calculus and Program Semantics* he jointly wrote with Carel Scholten.

The use of formalization and logic in programming brought Edsger into contact with a number of scientific issues he did not touch before. One of them was the history and foundations of logic. The problem was that Edsger was very much interested in using logic as an engineering instrument to facilitate thinking and reasoning, while typically a lot of what was published within mathematical logic dealt with foundations and concepts and properties of logics, and not so much with the ways logic could be used for solving practical problems.

Edsger immediately set out to work out his own presentation of logic. He developed an approach in which the main connective was equivalence and not implication, which is considered the most important concept in many logical systems. He indicated that one of his motivating reasons was that implication was often confused with causality and therefore it should be avoided. This is an interesting argument because causality is, certainly, a key concept in computer science and is definitively different from implication.

Edsger's approach to logic led him and Scholten to devote nearly 100 pages in their book to introduce logic and to present several proofs of small theorems that were more or less known to logicians. This was strongly criticized by logicians, in particular, by Egon Börger [1994] in his sharp review of Edsger's book.

However, even more important in the debate about Edsger's approach to logic is another question, namely: How does logic actually contribute to the development of software? The size of the software systems developed today is way beyond what was seen in the '80s and '90s. During the 1968 conference on software engineering, Edsger once said that compared with what was available in the '50s the machines used at the end of the '60s were giant computers and their programming caused giant problems. Today, when looking back, we have a completely different understanding of a giant computer. We are now considering computers available at the end of the '60s as tiny machines.

Edsger's lectures at Marktoberdorf from the '70s until the '90s show precisely his development. He started as someone predominantly interested in writing programs, also from a practical point of view, and moved in the direction of developing a foundational approach to programming including logics. As a result, he became deeply involved with logical questions—a topic which, to some extent, he was not educated for. In addition, this immediately led to his confrontation with logicians who did not understand his approach to logics, while he in turn did not appreciate logicians' approach to logics.

This is kind of a tragedy about Edsger's scientific history. Being at the beginning of the '70s perhaps the most well-known computer scientist, a man who—as Krzysztof Apt writes in Chapter 22—"carried computer science on his shoulders," he developed more and more into a person with a very specific, auto-didactic approach to logics as an instrument in programming, which—in the end—did not gain a lot of attention and recognition by the scientific communities, apart from a few logicians who deeply criticized Edsger's approach and a small group of followers who practiced his approach.

However, the most relevant question, still not adequately answered till today, is: What is the right methodology, the right approach to programming? We live now in times of high expectations from artificial intelligence, where, contrary to what Edsger was teaching, programming is mainly understood as machine learning, although it is quite clear that larger reliable programs cannot be developed that way. We live in times in which, in sharp contrast to what Edsger was teaching, agile programming and test-driven development approaches are being practiced.

Admittedly, the goto statements have disappeared. However, our discipline has not reached maturity, yet. We all are looking forward for further approaches to the methodology of program construction.

29.8 Edsger's Views about the Marktoberdorf Summer School

Edsger was very careful when giving lectures. It was important for him to give not presentations but lectures. He preferred to lecture just using a blackboard because he hated all the technical aids such as overhead transparencies or power point presentations.

In particular, he was always carefully thinking about how to address his audience and about the effect his lectures would have on the audience. In a number of EWDs he reported on the lectures he gave around the world and to what extent he considered them successful. It is pretty obvious how important it was for him to prepare these lectures carefully and also—after the lectures—to reflect on the effects they had on his audience.

This was also true about his Marktoberdorf lectures. In some of his EWDs he writes:

> Over the years The NATO Summer School in Marktoberdorf has become very definitely one of the highlights in my professional life as a lecturer: it is a real challenge! One has the opportunity to address an international audience of nearly a hundred young computing scientists, all of whom have come to Marktoberdorf with high expectations, and one has to try not to disappoint them, in spite of the fact that the expectations are as mixed as the audience itself and despite the handicap of possible language barriers.
> [. . .]
> Luckily, as the Summer School progresses, the majority of the participants discover that the official program of 56 lectures is not the main thing: the real exchange takes place when one is supposed to dine, when one is supposed to chat over a cup of coffee or a glass of beer, or when one is supposed to sleep. (I was pleased to hear that, in the dormitory, discussion often continued until three o'clock in the morning!) (EWD676 [Dijkstra 1978])

> It is fascinating to observe that the audience of the Marktoberdorf Summer School gets better each time. I can think of only two explanations. Firstly, as the Summer School gets more famous, the selection prior to application gets more effective. Secondly, with progressing internationalization—my apologies for that ugly word!—, fluency in English has improved to the extent that I observed little of the language barriers I remember from earlier Summer Schools: one heard, of course, all the marked accents, but this time even the French hardly clung together. The moral of the improving quality of the audience is clear: we have to be increasingly careful in the choice of our lecturers. (EWD895 [Dijkstra 1984b])

It was always remarkable to see the effect Edsger's lectures had on the participants of the summer school. We often scheduled Edsger's lecture as the first one in the morning with the result that literally all the participants showed up, even on days when there were beer parties the evening before and some of the participants would have preferred to sleep a bit longer. Edsger had a very direct way of addressing his audience, which was fascinating. He often let the audience take part in the way he thought about a problem and tried to solve it. In contrast to some other lecturers, he often showed how closely related were his personal style of attacking a problem and a way of working out a solution.

Being a part of the discussions at the end of the day, Edsger was very active. He always tried to come back to critical questions of the day and tried to give people additional material to think about.

29.9 A Line of Development

Edsger taught at Marktoberdorf for more than 30 years. It was fascinating to see how his teaching influenced the direction of the school and how his development was reflected in the lectures he presented there. In the beginning he was deeply concerned about an appropriate way of programming. It was a time when structured programming was advocated, mainly by him, Tony Hoare, Niklaus Wirth, and a couple of other researchers.

He relied very much on his experience gained from teaching programming in Algol 60, a subject on which he wrote a book [Dijkstra 1962], through which he acquired a deep understanding that programming was a major intellectual and engineering challenge.

Moreover, he was convinced that at that time most people did not understand the dimensions of this challenge. He was deeply depressed that at his university he did not get any support for making programming a first-class subject.

He was certainly inspired by the discussions held at the Garmisch-Partenkirchen conference on software engineering. Moreover, as he reports in his EWDs, he found a growing audience, in particular in Great Britain and the United States, for his ideas on programming. Giving lectures at many well-known universities and receiving the Turing Award brought him to a unique position. Unique in the sense that he soon became famous among computing scientists who were looking forward to seeing him and to listen to him. In addition, he could now use his lectures to aim to change the way people were constructing software. This created a unique context that was highly visible in his lectures and his contributions in Marktoberdorf.

With respect to Marktoberdorf he writes:

> Concluding Remarks.
> We have shown successive program versions, leading from the original problem statement to the final program. In our final program, the merging of these successive versions has been done by hand and the more abstract versions have been reduced to comment in the form of labels. For large programs this merging process itself becomes a major data processing task and I expect the growth of interactive program composition techniques in which the service of computers will be enlisted for the benefit of this process. Furthermore: at present the more abstract versions are only reflected as explanatory comment, inserted for human understanding. The origin of this is that we want at present a program formulated at a constant semantic level, viz. the level of the programming language. For the more abstract versions we have at present during run time no mechanical use. In future I expect the more abstract versions to be an integral part of the program. Finally, we have considered only a single line of programs leading from the problem statement to a working problem expressed at the desired semantic level. In future I expect that this single line will be extended to a more or less tree-structured class of programs in which is also room for alternatives, thus tying together program composition and program modification. (EWD241 [Dijkstra n.d.])

and

There is much confusion about that, and that confusion should not amaze us, for computers nowadays have so many different aspects:

- You can view computers primarily as industrial products, intended to be sold with profit. A burning question then becomes whether to advertise them with

Charlie Chaplin or with Mickey Mouse and a computing science curriculum should contain a course in Sales Promotion as a major component.

- You can view computers primarily as the main battlefield of international competition; in that case a computing science curriculum should mainly consist in courses on Security and Industrial Espionage.
- One day you can view automation as the motor of your economy, the other day you can view it as the greatest threat to the employment situation; so the main courses in our computing science curriculum should be on Economy and on Industrial Relations.
- In recognition of the fact that the new technology has its profoundest impact by adding a totally new dimension to the eternal philosophical questions Can machines think? and What is life?, we conclude that the major chairs in computing should be joint appointments between departments of Philosophy, of psychology, and of Biology.

(EWD896 [Dijkstra 1984c])

These thoughts go beyond what Edsger discussed in his lectures in Marktoberdorf. They show that already at that time he had some vision about how the discipline of software engineering might develop and might change the world. It is interesting that he viewed computers, or more precisely software, as the main battlefield of international competition. In addition, he mentioned security as an issue. This is precisely where we are today.

Moreover, he saw the gap between the science of programming as it was tackled by the universities and the practice of programming. In this way, he was talking about a "mismatch" that we can observe until today.

29.10 The End

In one of his last EWDs, Edsger looks back and reflects about the development of the topics he was studying: Starting with programming, the implementation of programming languages, operating systems, synchronization primitives, the development of algorithms for small intricate problems, the treatment of certain aspects of software systems such as self-stabilization, followed by structured programming and ways of developing programs and their correctness proofs hand-in-hand, which became his main concern in the '80s and '90s, together with looking at intricate problems that fascinated him.

Finally, in the '90s he became very much concerned about the proper way to do logic, heavily advocating the use of predicate algebra instead of predicate logic—an approach he talked about several times in Marktoberdorf.

Close to the end of his life he writes:

> These days there is so much obsession with application that, if the University is not careful, external forces, which do make the distinction, will drive a wedge between "theory" and "practice" and may try to banish the "theorists" to a ghetto of separate departments and separate buildings. A simple extrapolation will tell us that in due time the isolated practitioners will have little to apply; this is well-known, but has never prevented the financial mind from killing the goose that lays the golden eggs. The worst thing with institutes explicitly devoted to applied science is that they tend to become institutes of second-rate theory.
>
> [. . .]
>
> During my Burroughs years I changed very much as a mathematician. By the end of the 60s, R.W. Floyd and C.A.R. Hoare had shown us how to reason in principle about programs, but it took a few years before their findings penetrated through my skull. In the early 70s we studied Tony Hoare's correctness proof of the routine "FIND" and my friends still remember that at the end of that study I declared in disgust that "that meaningless ballet of symbols was not my cup of tea"; only gradually, my resistance to that kind of formula manipulation faded, until eventually, I began to love it.
>
> [...]
>
> Circumstances forced me to become a very different mathematician. I had invented predicate transformers as a formal tool for programming semantics and they served my purpose, but I lacked at the time the mathematical background needed to understand what I was doing, so much so that I published a paper with a major blunder. I was helped by C.S. Scholten, with whom I had worked since the 50s, and together we provided the mathematical background that had been lacking. In doing so we developed a much more calculational style of doing mathematics than we had ever practised before. I stayed away from where I had blundered, but that hurdle the younger and mathematically better equipped R.M. Dijkstra took in his stride after he had extended our apparatus from predicates to relations. (EWD1298 [Dijkstra 2000])

Looking back at Edsger and his magnificent contributions to the Marktoberdorf summer schools, and at the course of his thoughts over four decades, one can see two stages in his development.

As a superstar in his discipline, with deep insights and visions, he showed the way for many young computer scientists and participants in Marktoberdorf. His early discoveries were truly outstanding and his close interaction with his colleagues, in particular Tony Hoare, were highly inspiring. His work seems to have reached a peak in his contributions to structured programming, predicate transformers, and his book on the discipline of programming.

Afterwards he concentrated on understanding predicate transformers and their usage in program construction, more deeply and in more detail. But also then, he had a number of farsighted thoughts about the role and development of computing sciences. It would have been interesting to see how he would react to the explosion of software development in this century, which is far beyond we all had expected. Several problems Edsger was working on are still not completely solved. Hopefully, many of his early thoughts will help us to find our path into the future.

References

F. L. Bauer. 1976. Programming as an evolutionary process. In F. Bauer, E. Dijkstra, A. Ershov, M. Griffiths, C. Hoare, W. Wulf, and K. Samelson (Eds.), *Language Hierarchies and Interfaces*. Springer, Berlin, 153–182. DOI: https://doi.org/10.1007/3-540-07994-7_53.

H. Bekic. 1984. On the formal definition of programming languages. In *Programming Languages and Their Definition*, Vol. 177: Lecture Notes in Computer Science, Springer, 86–106. DOI: https://doi.org/10.1007/BFb0048941.

E. Börger. 1994. Book review E.W. Dijkstra, C.S. Scholten: Predicate Calculus and Program Semantics, Springer-Verlag, 1989. *Sci. Comput. Program.* 23, 4, 1–11. DOI: https://doi.org/10.1016/0167-6423(94)90002-7.

R. Burstall and J. Darlington. 1977. A transformation system for developing recursive programs. *J. ACM* 24, 44–67. DOI: https://doi.org/10.1145/321992.321996.

J. W. de Bakker. 1971. *Recursive Procedures*. Mathematisch Centrum.

E. W. Dijkstra. 1962. *Primer of Algol 60 Programming*. Academic Press.

E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inform.* 1, 115–138. DOI: https://doi.org/10.1007/BF00289519.

E. W. Dijkstra. August. 1978. Trip report E.W. Dijkstra, Marktoberdorf 24 July–6 August 1978. EWD676. http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD676.PDF.

E. W. Dijkstra. April. 1984a. Trip report E.W. Dijkstra, München, 12–14 April 1984. EWD885. http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD885.PDF.

E. W. Dijkstra. September. 1984b. Trip report E.W. Dijkstra, Marktoberdorf, 30 July–12 Aug. 1984. EWD895. http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD895.PDF.

E. W. Dijkstra. August. 1984c. On the nature of computing science. EWD896. http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD896.PDF.

E. W. Dijkstra. April. 2000. Under the spell of Leibniz's Dream. EWD1298. http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1298.PDF.

E. W. Dijkstra. n.d. Towards correct programs. EWD241. https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD241.PDF.

R. W. Floyd. 1967. Assigning meanings to programs. In *Proceedings of Symposium on Applied Mathematics*. American Mathematical Society.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10, 576–580. DOI: https://doi.org/10.1145/363235.363259.

Z. Manna. 1974. *Mathematical Theory of Computation*. McGraw-Hill, New York.

J. McCarthy. 1963. A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg (Eds.), *Computer Programming and Formal Systems*. North-Holland, Amsterdam, 33–70.

P. Naur and B. Randell. 1969. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, 7–11 October 1968. NATO Scientific Affairs Division, Brüssel.

D. S. Scott. 1972. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*. Springer-Verlag, Berlin, 97–136. DOI: https://doi.org/10.1007/BFb0073967.

30

The Edsger W. Dijkstra Archive

Hamilton Richards

The Edsger W. Dijkstra Archive[1] is an on-line repository of Edsger Dijkstra's otherwise unpublished manuscripts. It also contains video and audio recordings and links to obituaries and other articles about him, and a comprehensive BibTeX index of Edsger's works, including his publications.

Like most of us, Edsger always believed it a scientist's duty to maintain a lively correspondence with his scientific colleagues. To a greater extent than most of us, he put that conviction into practice. For over four decades, he mailed copies of his consecutively numbered technical notes, trip reports, insightful observations, and pungent commentaries to several dozen recipients in academia and industry. Because Edsger prefixed the manuscripts' numeric labels with his initials, they are known collectively as EWDs. Thanks to the ubiquity of the photocopier and the wide interest in Edsger's writings, the informal circulation of many of the EWDs eventually reached into the thousands. The original manuscripts, along with diaries, correspondence, photographs, and other papers, are housed at The Center for American History of The University of Texas at Austin.[2]

Although most of Edsger's publications began life as EWD manuscripts, the great majority of those manuscripts remained unpublished. They were inaccessible to many potential readers, and those who received copies were unable to cite them in their own work. To alleviate both of these problems, the university's Department of Computer Sciences collected PDF bitmap facsimiles of all of the known manuscripts and published them in a permanent web site.[3]

[1] https://www.cs.utexas.edu/∼EWD/.

[2] https://legacy.lib.utexas.edu/taro/utcah/00378/cah-00378.html.

[3] https://www.cs.utexas.edu/users/EWD/.

The Dijkstra Archive began life as a plan to provide the collection of EWD manuscripts on CD-ROMs to attendees at the Edsger W. Dijkstra Symposium in 2000. This celebration of Edsger's lifetime contributions to computing science was held at The University of Texas at Austin on May 12–13, 2000, a few months after his retirement on November 1, 1999. It began one day after his 70th birthday.

The symposium's organizers were Professor Jayadev Misra and Professor Stephen Keckler. Steve Keckler initiated the scanning of the EWDs by a local copying company, Speedway Copy. When about a third of the EWDs had been scanned, he asked me to take over. I accepted gladly, because I was a neighbor, colleague, and close friend of Edsger, whom I had known since he and I were both employed at Burroughs Corporation.

A few vignettes will convey the flavor of the efforts that went into the archive's creation.

Jay Misra enlisted the help of Wim H.J. Feijen, a long-time associate of Edsger's in Eindhoven, in rounding up copies of EWDs not in Edsger's possession in Austin. After handing over a list of 1033 titles, numbered EWD28 through EWD1284 (some numbers were assigned to documents that have not been found or were never issued), Wim wrote to Jay on January 28, 2000:

> Dear Jay,
> These days I am working hard on the earliest EWDs, trying to get that file as complete as possible. I have contacted Carel Scholten and CWI (the former Mathematical Center in Amsterdam), asking them for help for filling in some gaps. Browsing through my own archives, I sometimes encounter writings that never appeared as an EWD-note, but in which you and The Center of American History might be interested. One of those is a paper, entitled
>
> An Attempt to Unify the Constituent Concepts of Serial Program Execution,
>
> published in
>
> Symbolic Languages in Data Processing, Gordon & Breach, New York, 1962.
>
> Maybe you can ask your librarian to give you a copy.
>
> Also, I encounter EWDs of which I have no original, but just a poor copy. Thus far these are EWD35 and EWD75.
>
> ***
>
> The reason I write this to you is to report all irregularities, so that you can take notice of them. I am NOT following your advice to send the originals

> without first having made copies. In the first batch—which I hope to mail next week (to RCC[4])—you will find, besides originals, copies for the purpose of scanning. I will keep you informed.
>
> Regards,
>
> Wim
>
> PS Don't get nervous. The first 250, say, EWDs require the bulk the work. Have you started scanning those from EWD250 onwards already?

Wim enlisted the aid of another significant contributor, Professor Krzysztof Apt, now CWI Fellow at Centrum Wiskunde & Informatica (formerly the Mathematisch Centrum) in Amsterdam. Krzysztof found 34 technical reports that Edsger wrote while he was working at the Mathematisch Centrum. Written before the start of the EWD series, these are listed in the archive under MC Reports.

A BibTeX index of EWDs[5] was provided to us by Rajeev Joshi[6], then a student of Jay's and a member of the Austin Tuesday Afternoon Club, who had independently compiled a database of EWDs.

The bitmap scans required checking for images that were missing, misaligned, or otherwise deficient. Most of the problems afflicted the early EWDs because these were the oldest documents and the scanning process was still being fine-tuned. To support the checking, I set up a makeshift website containing the PDF scans, along with a grading scheme for reporting problems.

Scan checking was done by several of Edsger's colleagues. Rob Hoogerwoord checked Edsger's dissertation, and the EWDs were checked by Steve Keckler (500–699), Rajeev Joshi (700–799), Lex Bijlsma[7] (800–899), Jay Misra (900–950), and Pamela B. Lawhead[8] (1000–1199). The rest of the scans were checked by Wim Feijen. Some of the defects uncovered in the checking required rescanning, but many others were corrected using Photoshop.

Concurrently with the scanning, checking, and searching for EWDs, we conducted an investigation of copyright issues. The archive was never intended to contain actual publications, but many of the EWD manuscripts were precursors of published works. Were manuscripts and typescripts that led to publications subject to the publishers' copyrights in the published versions? We consulted with the university's copyright office:

[4] Robbie Creek Cove, the street in Austin on which the Dijkstras were living.

[5] https://www.cs.utexas.edu/EWD/indexBibTeX.html.

[6] Principal Applied Scientist, Automated Reasoning Group at Amazon Web Services.

[7] Professor Emeritus at the Faculteit Management, Science & Technology, Open University, Heerlen, The Netherlands.

[8] Assistant Professor, Computer and Information Science, The University of Mississippi.

> HR: Does a publisher's copyright in a published work extend to the manuscript from which the publication was derived?
> UT Office: NO, unless the publisher specifically included some kind of language claiming that the assignment to it included copyright in the manuscript.
> HR: Is the publisher's claim to copyright protection weakened by differences between the manuscript and the publication resulting from editing? In other words, how do we determine the dividing line between an idea and an expression of an idea?
> UT Office: The dividing line you are looking for is between the pre-publication expression and the published expression. [. . .]
> HR: In cases where we're not sure, is it better to play it safe and ask permission?
> UT Office: Yes.

Following this advice, we sought permission for all of the manuscripts that we knew had resulted in publications. The publishers' responsiveness varied widely. The quickest response came from Springer-Verlag, which not only granted no-fee permissions for our first batch of requests but responded equally graciously as we discovered more EWDs that had given rise to S-V publications.

Academic Press initially declined resolutely to waive its $110 fee for permission to display a single manuscript for only two years. The solution, as related by Edsger:

> Ria[9] said "Why don't you call Otto ter Haar?" I know Otto since 1948, and since then he became the boss of Elsevier. I did so, and he immediately said to contact Pieter Bolman in San Diego, adding "He should be very cooperative for we are in the process of buying them."

Pieter Bolman, the president of Academic Press, kindly waived the fee and the time limit.

The last holdout was Prentice-Hall International, which was unable to determine whether the copyright in one of the EWDs was held by them. At Edsger's suggestion I contacted Tony Hoare, the editor of the volume containing Edsger's contribution. He informed me that the copyright was held by the Royal Society, which granted permission immediately.

Upon receipt of permission for a manuscript, we added it to the growing CD-ROM collection, preceded by a notice that permission to include it in the CD-ROM and the archive website had been granted by the publisher.

[9] Edsger's wife, Maria Dijkstra-Debets.

The copyright problems combined with other production delays to keep the CD-ROM from being finished in time for the May symposium. Copies were finally mailed to the symposium attendees in November.

We pondered whether EWDs marked "Confidential," or intentionally withheld from distribution, should be posted. Edsger had asked EWD1127's recipients to destroy their copies, so it was not added to the archive, but after initially recommending against posting EWDs 1018, 1035, and 1110, Jay Misra relented:

> On further reflection, I agree with you. They declassify documents after a decent interval, and 18 years is long enough.

The archive's first major update began in August 2002 with a suggestion by former Ph.D. student John Adair:

> Although his handwriting and the care with which they are written provide much of their charm, I was thinking that it might be nice to have text (html) versions of them as well. If there's ever a project to convert them and you're looking for volunteers, please shoot me an email at ⟨URL⟩

The main advantage of textual versions would be their searchability. The collection amounted to around 7900 pages, so transcribing them would be a massive task. We considered OCR, but quickly determined that although it would probably work well enough on typescript, it would be useless for handwritten text, even text penned in Edsger's elegant hand. A large majority of the EWDs are handwritten and would therefore have to be transcribed manually. Another suggestion was to read the handwritten EWDs aloud into a speech recognizer, but that idea went nowhere.

Of course, the presence of the EWDs on the web made the transcription task ideal for splitting up: Each document constitutes an independent transcription task, accessible to any volunteer with Internet access. The only requirement would be a system for coordinating volunteer transcribers to ensure that each document would be transcribed at most once.

At first the obvious medium for transcriptions appeared to be LaTeX, which "everybody knows." Alternatives we considered included XML, MathML, and HTML, the last of which seemed a poor third because

> Right now HTML support for math and formulae is pretty weak. Pretty much the best you can do is to make the formulae into images, or use some sort of plug-in for them.

After a hiatus occasioned by the demands of the academic year, the discussion resumed in May 2003, when John Adair and I were joined by Ken Dyck, who had on his own initiative transcribed several EWDs into LaTeX.

Continuing the hunt for the best medium, I suggested plain text, arguing that

> That may seem like a highly retrograde proposal, but as I see it, we're not trying to replace Edsger's handwriting. Whenever I see one of his documents typeset, no matter how beautifully, I can't help thinking that something is missing; our real goal is to augment the scanned documents, to make them searchable by everybody and accessible to the visually impaired.
>
> Once a search has succeeded, I expect a sighted searcher would turn to the handwritten original. If s/he wanted to extract a quote, plain text would be easier to copy and paste than any formatted alternative.
>
> Visually impaired readers will presumably be using software that "speaks" text. I don't know much about that software, but I suppose that it would work better in the absence of formatting codes.
>
> The main drawback of plain text would seem to be its inability to deal with mathematical formulas. Edsger's notation, however, is typographically about as unchallenging as they come—he designed it to be easy to type on an ordinary typewriter, so it uses very few exotic characters, and tends to be quite one-dimensional. Be that as it may, sighted readers would naturally turn to the original handwritten formulas, and visually impaired ones would be better served by an ascii approximation than by elaborate mathematical typesetting.

We eventually settled on HTML with Unicode, after John Adair used it to produce a nice transcription of EWD1313.[10] For volunteer transcribers, we proposed to offer three levels of transcription:

0. text only, with formulas indicated by [formula], for folks who don't want to mess with `&nbsp;x&#x2193;` and so on.
1. text plus formulas
2. text plus formulas plus html.

We would assure any transcriber who stopped short of Level 2 that they should feel comfortable that they had made a major contribution, and that someone else would be glad to add the upper-level details.

In the archive website, I set up an index of the EWDs comprising an index page for each hundred from 0 to 1300, to make it easy to see which EWDs are missing.

[10] https://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1313.html.

Also, I added a separate index page for the non-EWD documents. Each index page consists of an HTML table with one row for each document. A document's row contains its EWD number, the size of its PDF file, and its title (with an embedded link to the PDF). For transcribed documents, we added a link to its transcription, which makes it easy to spot candidates for transcription.

To ensure that no document would be transcribed twice, I set up a separate index of transcriptions[11], indicating for each one whether it was completed or in progress.

One of the transcribers, Martin P.M. van der Burgt, devised a process for producing transcripts automatically. Although their markup was incomplete, we believed they would serve a useful purpose by virtue of their searchability and their accessibility to text-reading software. They are marked in the transcription index as "incomplete," and the plan is to replace them with fully marked-up versions as time permits.

A significant aid in recruiting transcribers was the list we maintained of site visitors who asked to be notified when the site was updated. The notifications included this text:

> By asking to be notified when the Edsger W. Dijkstra site is updated, you've indicated that you find Edsger's writings interesting. Now there's a way to express your interest in a way that will make the site more useful and accessible—by joining in an effort to transcribe the EWDs to searchable text. You can read all about it at http://www.cs.utexas.edu/users/EWD/transcriptions/invitation.html.

Several visitors suggested that the EWDs should be typeset and published in book form. A sample dialog:

> Visitor: I have been reading through your amazing Dijkstra archive, and wondered if there are any plans to publish the EWDs in printed form, similar to what Knuth is doing with his "Selected Papers" series, where he is re-typesetting his old papers using TEX, grouping them by subject into 8 volumes, and publishing them through CSLI.
> HR: If there are any such plans, no one has mentioned them to me. That would be a very substantial undertaking.
> Visitor: Of course, Knuth's old papers weren't beautifully hand-written, but the EWDs might still be more suitable for publishing if they are typeset, particularly those with mathematical content.

[11] https://www.cs.utexas.edu/users/EWD/transcriptions/transcriptions.html.

> HR: My own feeling is that the cost of typesetting the EWDs would far outweigh any gain in “suitability for publishing”. Indeed, it’s not clear to me that there would be any such gain, especially as Dijkstra is no longer available to proofread the results. Such a project could not be undertaken without substantial funding, and I can’t think of any convincing arguments in favor of it.
>
> The archive of PDF images is in the spirit of the samizdat-style distribution of the original EWDs. The current transcription project is intended only to make it easier to locate the EWDs in which some word or phrase occurs, and to broaden the audience to include people who are blind.
>
> Visitor: OK, thanks for the reply. I have no idea how many copies would be sold, but somehow the economics work for Knuth.
>
> HR: In 1982 Springer did publish quite a few EWDs in
>
>> *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.
>
> I don’t know how the economics worked for Dijkstra, but he wasn’t tempted to do it again, perhaps partly because of the tremendous amount of essentially clerical work it entailed. In those days, the payoff was the greatly increased availability of the EWDs; nowadays that motivation has been erased by the Internet.
>
> And of course quite a number of the EWDs were drafts of journal articles and book chapters. Those were presumably the ones most worth publishing, and they would require securing permission from their publishers. That exercise, which I undertook for the web site, is one I’m eager not to repeat.

At this writing, the 111 transcribers have produced 823 transcriptions, which is 74% of the 1,116 PDF scans. Both increased rapidly in the early years, and reached a plateau in 2015 (Figure 30.1).

Long after Edsger’s death, EWDs continued to be added to the archive. In 2013, for instance, Erik Geelen, Academic Heritage Specialist at the Eindhoven University of Technology, sent us scans of three EWDs—301, 356, and 360—which were missing from the archive. They had turned up in the collection of the late Professor Martin Rem, who had gained his Ph.D. under Edsger’s supervision.

Some professional archivists argued that the transcriptions should resemble the originals in all possible respects—page breaks, line breaks, and so on. My response was that for every transcription the original scan is one click away, readily available to anyone who cares about the format of the original. The transcriptions’

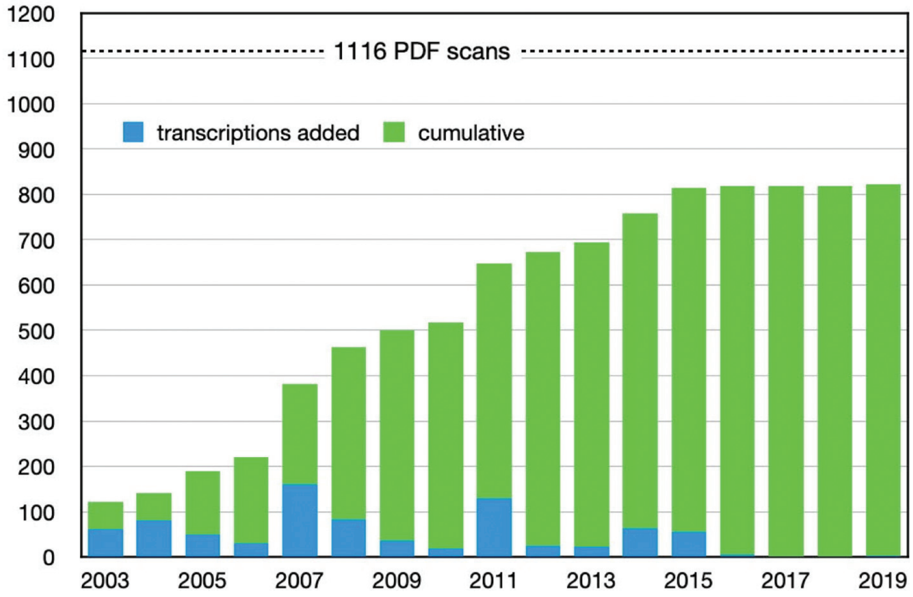


Figure 30.1 Transcriptions of EWDs and other documents.

purpose is not to substitute for the original, but merely to make the EWDs searchable.

Several visitors offered to translate into English the EWDs and other manuscripts written in Dutch, and indeed several translations[12] were added to the archive. Being unable to read any Dutch myself, I asked Edsger's closest collaborator, Wim Feijen, for assistance in evaluating translations. His response:

> Translation is an activity that requires a highly developed skill in at least two languages, and the problem with Edsger is that his mastery of Dutch is far beyond what an average translator can grasp.
> In your case, I would stop chasing opportunities for translating Edsger's Dutch writings, no matter how you would like doing so . . ..

Wim's recommendation was reinforced by Edsger's confession, after he attempted to translate EWD724[13] that "this confrontation with the translator's problems was an educational experience," and (referring to a colleague who assisted in the translation) "neither of us is fully satisfied" with the result.

[12] https://www.cs.utexas.edu/users/EWD/translations/translations.html.

[13] https://www.cs.utexas.edu/users/EWD/transcriptions/EWD07xx/EWD724E.html.

Taking Wim's advice, I regretfully brought the effort to translate any more Dutch EWDs into English to a halt.

The curiosity that untranslated documents arouse in non-speakers of Dutch can in principle be relieved by summaries,[14] which were suggested by a transcriber, Günter Rote. Of the 30 summaries contributed so far, five summarize documents in Dutch.

Toward the end of 2018, my feeling that the Dijkstra Archive needed a new curator could be ignored no longer. My calls for a volunteer among the UTCS faculty drew no responses, so I suggested that a suitable successor for me would be Rutger M. Dijkstra. Edsger's younger son, Rutger is a graduate in computing science from the University of Groningen.

Rutger was willing, and by the end of 2019 all the necessary approvals had been obtained. In January 2020, Rutger became the archive's official curator.

[14] https://www.cs.utexas.edu/users/EWD/Summaries/Summaries.html.

Edsger W. Dijkstra: Biographical Information

Krzysztof R. Apt

Edsger Wybe Dijkstra
Born May 11, 1930, in Rotterdam. Died August 6, 2002, in Nuenen.

Education

Master of Physics, University of Leiden (1956)
Ph.D. thesis, University of Amsterdam, "Communication with an Automatic Computer"; supervisor: Prof. A. van Wijngaarden (1959)

Positions held

Programmer, Mathematisch Centrum, Amsterdam, the Netherlands (1952–1962)
Professor of Mathematics, Technical University of Eindhoven, the Netherlands, (1962–1984)
Research fellow, Burroughs Corporation, Nuenen, the Netherlands (1973–1984)
Schlumberger Centennial Chair in Computer Sciences, University of Texas at Austin, USA (1984–1999)

Awards and Honors

Member of the Royal Netherlands Academy of Arts and Sciences (1971)
Distinguished Fellow of the British Computer Society (1971)
ACM Turing Award (1972)
IEEE Computer Society Harry H. Goode Memorial Award (1974)
Foreign Honorary member of the American Academy of Arts and Sciences (1975)
Doctor of Science Honoris Causa, Queen's University of Belfast (1976)
Computer Pioneer Award, IEEE Computer Society (1982)

ACM/SIGCSE Award for Outstanding Contributions to Computer Science Education (1989)
ACM Fellow (1994)
Honorary doctorate, Athens University, Greece (2001)
ACM PODC Influential Paper Award (2002) (The award was renamed in 2003 to Edsger W. Dijkstra Prize in Distributed Computing)
C&C Prize, The NEC C&C Foundation, Japan (2002)

Published Works

1959

A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 269–271.

Communication with an Automatic Computer. Ph.D. thesis. University of Amsterdam. http://www.cs.utexas.edu/users/EWD/PhDthesis/PhDthesis.PDF.

(with W. Heise, A. J. Perlis, and K. Samelson). ALGOL sub-committee report—Extensions. *Commun. ACM* 2, 9, 24.

1960

Recursive programming. *Numer. Math.* 2, 312–318.

1961

Letter to the editor: Defense of ALGOL 60. *Commun. ACM* 4, 11, 502–503.

1962

Some meditations on advanced programming. In *IFIP Congress*. North-Holland, 535–538.

Operating experience with ALGOL 60. *Comput. J.* 5, 2, 125–127.

An attempt to unify the constituent concepts of serial program execution. In *Proceedings of the Symposium Symbolic Languages in Data Processing*. Stichting Mathematisch Centrum, Vol. 65, 237–252. Gordon and Breach. MR46/62. https://www.cs.utexas.edu/users/EWD/MCReps/MR46.PDF.

Primer of Algol 60 Programming. Academic Press.

1964

Some comments on the aims of MIRFAC. *Commun. ACM* 7, 3, 190.

1965

Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9, 569.

Programming considered as a human activity. In *Proceedings of the IFIP Congress*, Vol. 65, 213–217.

1968

Go to statement considered harmful. *Commun. ACM* 11, 3, 147–148. Letter to the Editor.

The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5, 341–346. Reprinted in P. Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming.* Springer, 139–152.

The go to statement reconsidered. A Reply by E. W. Dijkstra. *Commun. ACM* 11, 8, 538, 541. Letter to the Editor.

Cooperating sequential processes. In F. Genuys (Ed.), *Programming Languages: NATO Advanced Study Institute*. Academic Press, 43–112. Reprinted in P. Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming*, Springer, 65–138.

A constructive approach to the problem of program correctness. *BIT Numer. Math.* 8, 174–186.

1969

Complexity controlled by hierarchical ordering of function and variability. In *Software Engineering*. NATO Science Committee, 181–185.

1970

Structured programming. In *Software Engineering Techniques*, NATO Science Committee, 65–69.

1971

Hierarchical ordering of sequential processes. *Acta Inform.* 1, 115–138. Reprinted in P. Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming*. Springer, 198–227.

Concern for correctness as a guiding principle for program construction. In *Infotech State of the Art Report 1: The Fourth Generation*. Infotech, 357–367.

1972

A class of allocation strategies inducing bounded delays only. In *AFIPS Spring Joint Computing Conference*, Vol. 40: *AFIPS Conference Proceedings*. ACM, 933–936.

Hierarchical ordering of sequential processes. In C. A. R. Hoare and R. H. Perrot (Eds.), *Operating Systems Techniques*. Academic Press, 72–93.

Information streams sharing a finite buffer. *Inf. Process. Lett.* 1, 179–180.

The humble programmer. *Commun. ACM* 15, 10, 859–866.

(with O. Dahl and C. A. R. Hoare). *Structured programming*, Vol. 8 A.P.I.C. Studies in Data Processing. Academic Press. ISBN 978-0-12-200550-3.

1974

Programming as a discipline of mathematical nature. *Am. Math. Mon.* 81, 6, 608–612.

Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11, 643–644.

1975

On the teaching of programming, i.e. on the teaching of thinking. In *Language Hierarchies and Interfaces*, Vol. 46: Lecture Notes in Computer Science. Springer, 1–10.

Guarded commands, nondeterminacy and a calculus for the derivation of programs. In *Language Hierarchies and Interfaces*, Vol. 46: Lecture Notes in Computer Science. Springer, 111–124.

A time-wise hierarchy imposed upon the use of a two-level store. In *Language Hierarchies and Interfaces*, Vol. 46: Lecture Notes in Computer Science. Springer, 345–357.

Craftsman or scientist. In *Data: Its Use, Organization and Management: ACM Pacific*. ACM Press, 217–223.

Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8, 453–457.

Correctness concerns and, among other things, why they are resented. *ACM SIGPLAN Not.* 10, 6, 546–550.

(with L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens). On-the-fly garbage collection: An exercise in cooperation. In *Language Hierarchies and Interfaces*, Vol. 46: Lecture Notes in Computer Science. Springer, 43–56.

1976

A Discipline of Programming. Prentice-Hall.

Formal techniques and sizeable programs. In *ECI*, Vol. 44: Lecture Notes in Computer Science. Springer, 225–235.

The effective arrangement of logical systems. In *Proceedings of the 5th Symposium Mathematical Foundations of Computer Science*, Vol. 45: Lecture Notes in Computer Science. Springer, 39–51.

On a gauntlet thrown by David Gries. *Acta Inform.* 6, 357–359.

1977

Programming: From craft to scientific discipline. In *International Computing Symposium.* North-Holland, 23–30.

1978

A more formal treatment of a less simple example. In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 2–20.

Stationary behaviour of some ternary networks. In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 21–23.

Finding the correctness proof of a concurrent program. In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 24–34.

On the interplay between mathematics and programming. In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 35–46.

A theorem about odd powers of odd integers. In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 47–48.

In honour of Fibonacci. In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 49–50.

On the foolishness of “natural language programming.” In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 51–53.

Program inversion. In *Program Construction: International Summer School*, Vol. 69: Lecture Notes in Computer Science. Springer, 54–57.

DoD-I: The summing up. *ACM SIGPLAN Not.* 13, 7, 21–26.

Finding the correctness proof of a concurrent program. In *Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium*, Vol. 64: Lecture Notes in Computer Science. Springer, 31–38.

Finding the correctness proof of a concurrent program. *Proc. Koninklijke Nederlandse Akademie van Wetenschappen A* 81, 2, 207–215.

(with L. Lamport, A. Martin, C. Scholten, and E. Steffens). On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11, 966–975.

1979

Software engineering: As it should be. In *Proceedings of the 4th International Conference on Software Engineering*. IEEE Computer Society, 442–448.

Verkavelde berekeningen, hun mogelijkheden en moeilijkheden. *Proc. Koninklijke Nederlandse Akademie van Wetenschappen 88*, 3, 21–23. In Dutch.

My hopes of computing science. In *Proc. 4th Int. Conf. on Software Engineering, Munich*. IEEE, 442–448.

1980

A programmer's early memories. In J. Howlett, N. Metropolis, and G.-C. Rota (Eds.), *A History of Computing in the Twentieth Century, A Collection of Essays*. Academic Press, 563–573.

Some beautiful arguments using mathematical induction. *Acta Inform.* 13, 1–8.

(with C. S. Scholten). Termination detection for diffusing computations. *Inf. Process. Lett*. 11, 1, 1–4.

1981

A word of welcome. *Sci. Comput. Program.* 1, 1–2, 3–4.

1982

Smoothsort, an alternative for sorting in situ. *Sci. Comput. Program.* 1, 3, 223–233.

How do we tell truths that might hurt? *ACM SIGPLAN Not.* 17, 5, 13–15.

Mathematical induction and computing science. *Nieuw Archief voor Wiskunde* 30, 2, 117–123.

De software crisis, ontstaan en hardnekkigheid. *De computer krant* 3, 20. In Dutch.

Selected Writings on Computing: A Personal Perspective. Texts and Monographs in Computer Science. Springer-Verlag.

(with A. J. M. van Gasteren). An introduction to three algorithms for sorting in situ. *Inf. Process. Lett*. 15, 3, 129–134.

1983

The fruits of misunderstanding. *Elektron. Rechenanlagen* 25, 6, 10–13.

(with W. H. J. Feijen and A. J. M. van Gasteren). Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett*. 16, 217–219.

1984

Invariance and nondeterminacy. Philos. Trans. R. Soc. A Math. Phys. Sci. 312, 1522, 491–499.

(with W. H. J. Feijen). *Een methode van programmeren*. Academic Service. In Dutch.

1985

Invariance and nondeterminacy. In *Mathematical Logic and Programming Languages.* Prentice-Hall, 157–164.

1986

A belated proof of self-stabilization. *Distrib. Comput.* 1, 5–6.

On a cultural gap. *Math. Intell.* 8, 1, 48–52.

A heuristic explanation of Batchers's Baffler. *Sci. Comput. Program.* 9, 213–220.

(with A. J. M. van Gasteren). A simple fixpoint argument without the restriction to continuity. *Acta Inform.* 23, 1, 1–7.

1987

Position paper on "fairness." *Softw. Eng. Notes* 13, 2, 18–20. EWD398. https://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1013.PDF.

1988

A new science, from birth to maturity. In *ETH Zürich: 20-Year Anniversary of Institut für Informatik.* ETH Zürich, Institut für Informatik.

On binary operators and their derived relations. *BIT Numer. Math.* 28, 378–382.

(with W. H. J. Feijen). *A method of programming*. Addison-Wesley. Translation of *Een methode van programmeren*.

1989

(with W. H. J. Feijen). The linear search revisited. *Struct. Program*. 10, 1, 5–9.

1990

Formal Development of Programs and Proofs. Addison-Wesley (Editor).

The derivation of a proof by J.C.S.P. van der Woude. In *Formal Development Programs and Proofs*. Addison-Wesley, chapter 16, 201–207.

Fillers at the YoP Institute. In *Formal Development Programs and Proofs*. Addison-Wesley, chapter 17, 209–227.

(with C. S. Scholten). *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag.

Making a fair roulette from a possibly biased coin. *Inf. Process. Lett*. 36, 193.

1992

On the economy of doing mathematics. In *International Conference on the Mathematics of Program Construction*, Vol. 669: Lecture Notes in Computer Science. Springer, 2–10.

The unification of three calculi. In *Proceedings of the NATO Advanced Study Institute on Program Design Calculi*, Vol. 118: NATO ASI Series. Springer, 197–231.

1996

Fibonacci and the greatest common divisor. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*. Springer, 7–10.

The balance and the coins. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*. Springer, 11–13.

Bulterman's theorem on shortest trees. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*. Springer, 15–16.

A prime is in at most 1 way the sum of 2 squares. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*. Springer, 17–20.

A bagatelle on Euclid's algorithm. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*. Springer, 21–23.

On two equations that have the same extreme solution. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*. Springer, 25–26.

An alternative of the ETAC to EWD1163. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design.* Springer, 27–28.

The argument about the arithmetic mean and the geometric mean, heuristics included. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*. Springer, 29–32.

2000

On the transitive closure of a wellfounded relation. In L. Böszörményi, J. Gutknecht, and G. Pomberger (Eds.), *The School of Niklaus Wirth, "The Art of Simplicity."* dpunkt.verlag and Academic Press. Copublication with Morgan-Kaufmann, 31–38.

The end of computing science? *Commun. ACM* 44, 3, 92.

2001

Under the spell of Leibniz's dream. *Inf. Process. Lett.* 77, 2–4, 53–61.

(with J. Misra). Designing a calculational proof of Cantor's theorem. *Am. Math. Mon.* 108, 5, 440–443.

2002

EWD1300: The notational conventions I adopted, and why. *Form. Asp. Comput.* 14, 2, 99–107.

EWD1308: What led to “Notes on Structured Programming.” In *Software Pioneers*, Springer Berlin, 340–346.

A synthesis emerging? In P. B. Hansen (Ed.), *The Origin of Concurrent Programming*. Springer, 397–412. 2005

2005

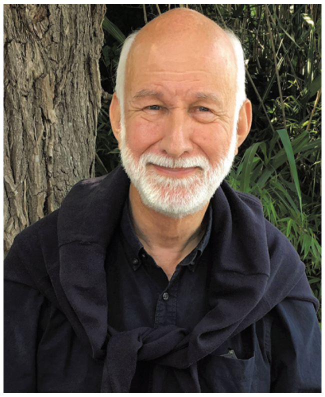
My recollections of operating system design. *ACM SIGOPS Oper. Syst. Rev.* 39, 2, 4–40.

2009

On the theorem of Pythagoras. *Nieuw Archief voor Wiskunde* 5, 10, 95–99.

Authors' Biographies

Krzysztof R. Apt



Krzysztof R. Apt is a Fellow at CWI (Centre Mathematics and Computer Science) in Amsterdam and Affiliated Professor at the University of Warsaw. He is also Professor Emeritus at the University of Amsterdam. He earned his Ph.D. degree in mathematics from the Polish Academy of Sciences in Warsaw in 1974. During his scientific career, he held tenure positions in Poland, France, the USA, and the Netherlands.

Apt published four books and several articles in computer science, mathematical logic, and, more recently, theoretical economics. In computer science his research interests have included program correctness and semantics, use of logic as a programming language, design of programming languages, distributed computing, and algorithmic game theory. For the past 20 years he has been involved in a number of initiatives aiming at open access to scientific publications.

He is a member of Academia Europaea, the founder and first Editor-in-Chief of the *ACM Transactions on Computational Logic*, and past president of the Association for Logic Programming. Together with Tony Hoare, he is the editor of this volume.

Tony Hoare

Tony Hoare was on the faculty of the Queen's University of Belfast from 1968 until 1977. He moved to Oxford University as Professor of Computation in 1977, where he remained until his retirement from academia in 1999. Shortly thereafter he joined the Microsoft Research Laboratory in Cambridge (UK).

His research has spanned several aspects of programming including design of data structures and programming languages, program verification, and concurrency. He invented Quicksort, conceived Hoare logic, proposed (jointly with Per Brinch Hansen) the concept of a monitor, and introduced *Communicating Sequential Processes* both as a language for distributed programming and, later, as a formalism to reason about concurrency and nondeterminism. His more recent work is concerned with the Unifying Theories of Programming.

Hoare received the Turing Award in 1980, the Harry H. Goode Memorial Award in 1981, the Kyoto Prize in 2000, and the IEEE John von Neumann Medal in 2011. In 2000, he was knighted by the British Queen for services to education and computer science. He holds honorary doctorates from several universities and is a fellow or foreign member of various learned societies, including the UK Royal Society, the UK Royal Academy of Engineering, the US National Academy of Sciences, the US National Academy of Engineering, and the Computer History Museum. Together with Krzysztof R. Apt, he is the editor of this volume.

Manfred Broy

Manfred Broy received his diploma in mathematics and informatics in 1976 and finished his Ph.D. at the Technical University of Munich, Germany, under the supervision of Friedrich L. Bauer in 1980. After his habilitation in 1982, he became a professor for informatics at the University of Passau, Germany, in 1983, and the founding dean of the faculty of computer science and mathematics. Since 1989, he has worked as a full professor at the faculty of computer science of the Technical University of Munich.

Manfred Broy has published more than 500 papers and received a number of honors: among them several memberships of academies such as acatech (National Academy of Science and Engineering) and the German Academy of Natural Sciences (Leopoldina).

E. Allen Emerson

E. Allen Emerson received in 1976 his Bachelor of Science degree in mathematics from the University of Texas at Austin and in 1981 his Ph.D. degree in applied mathematics from Harvard University. Since 1981, he has been working at the University of Texas at Austin, where he is Professor and Regents Chair Emeritus.

He is coinventor and codeveloper of model checking, an algorithmic method of verifying finite-state concurrent programs. For his work he became in 1998 a joint recipient of the ACM Kanellakis Prize, followed in 1999 by the CMU Newell Medal. In 2006, he obtained the IEEE LICS'06 Test-of-Time Award.

In 2007, he received, jointly with Edmund M. Clarke and Joseph Sifakis, the ACM Turing Award "for their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries."

Robert van de Geijn and Maggie Myers

Robert van de Geijn received his Ph.D. in Applied Mathematics in 1987 and **Maggie Myers** hers in Statistics in 1988 from the Department of Mathematics at the University of Maryland. During 34 years on the faculty of The University of Texas at Austin, they were both affiliated with the Department of Computer Science, the Department of Statistics and Data Sciences, and the Oden Institute for Computational Engineering and Sciences. Maggie was also affiliated with the Department of Mathematics and The Charles A. Dana Center.

They retired in 2021, after which they have continued to be involved in research and outreach through the Science of High-Performance Computing group. Together, they have developed four Massive Open Online Courses that are available on the edX platform: on undergraduate and graduate linear algebra, programming for correctness, and programming for high performance.

David Gries

David Gries received his Ph.D. (actually Dr. rer. nat.) in Math from the Munich Institute of Technology in 1966. After 3 years at Stanford, he joined the CS Department at Cornell University, where he has been ever since except for a brief sojourn at the University of Georgia. He is now Professor Emeritus.

Gries was the major implementor of the ALCOR-ILLINOIS Algol 60 compiler, working closely with Manfred Paul and Hans Rüdiger Wiehle, who designed it. In 1971, he published the first textbook on the subject, *Compiler Construction for Digital Computers*.

A major research area has been programming methodology and related topics—semantics, practical logic, and so on. His work has resulted in several major books on the science of programming and a logical approach to discrete math.

Gries has an Honorary Doctor of Science from Miami University, Ohio, and an Honorary Doctor of Laws from Daniel Webster College, New Hampshire. He received four international awards for education: the ACM Karlstrom Award, the IEEE Taylor Booth Education Award, the ACM-SIGCSE Education Award, and the AFIPS Education Award. An article with Susan Owicki on interference freeness won the ACM Programming Systems and Languages Paper Award in 1977.

Reiner Hähnle

Reiner Hähnle received a Diploma in Computer Science from Karlsruhe University (now KIT) and a Ph.D. in 1992. In 1997, he habilitated at Technical University of Vienna and joined the faculty of Chalmers University of Technology in Gothenburg in 2000. In 2002, he became Full Professor at Chalmers; in 2011, he moved to Technical University of Darmstadt, where he holds the Chair of Software Engineering.

He has made contributions to proof theory, automated theorem proving, software verification, and distributed programming languages. In 1999, he initiated the still ongoing KeY project on deductive software verification. Since 2009, he has been involved in the development of the distributed active object language ABS.

He has published over two hundred technical papers and is author or (co-)editor of 11 books. He cofounded the Tableaux and IJCAR conference series, and he was a founding trustee of Federated Logic Conferences, Inc.

Ted Herman

Ted Herman is a Professor Emeritus at the University of Iowa. His doctoral work, at UT Austin under the supervision of Mohamed Gouda, explored topics in self-stabilization. Subsequently, he was a founding member of the steering committee of the Workshop on Self-Stabilizing Systems, and later the Symposium on Stabilization, Safety, and Security of Distributed Systems. During the early 2000s, his research focused on the emerging area of wireless sensor networks. His current research applies sensor networks to computational epidemiology.

Butler Lampson

Butler Lampson was on the faculty at Berkeley and then at the Computer Science Laboratory at Xerox PARC and at Digital Equipment Corporation's Systems Research Center. Since 1995, he has been a Technical Fellow at Microsoft Research and an Adjunct Professor of Computer Science and Electrical Engineering at MIT.

He has worked on computer architecture, local area networks, raster printers, page description languages, operating systems, remote procedure call, programming languages and their semantics, programming in the large, fault-tolerant computing, transaction processing, computer security, WHSIWYG editors, and tablet computers. He was one of the designers of the SDS 940 time-sharing system, the Alto personal distributed computing system, the Xerox 9700 laser printer, two-phase commit protocols, the Autonet LAN, the SDSI/SPKI system for network security, the Microsoft Tablet PC software, the Microsoft Palladium high-assurance stack, and several programming languages including Mesa and Euclid.

He received an A.B. from Harvard University, a Ph.D. in EECS from the University of California at Berkeley, and honorary Sc.D.'s from the Eidgenössische Technische Hochschule, Zürich, and the University of Bologna. He is a member of the US National Academy of Sciences and National Academy of Engineering, a Foreign Member of the Royal Society, and a Fellow of the Association for Computing Machinery, the American Academy of Arts and Sciences, and the Computer History Museum. He received the ACM Software Systems Award in 1984 for his work on the Alto, the IEEE Computer Pioneer award in 1996, the National Computer Systems Security Award in 1998, the IEEE von Neumann Medal in 2001, the Turing Award in 1992, and the National Academy of Engineering's Draper Prize in 2004.

At Microsoft, he has worked on anti-piracy, security, fault-tolerance, and user interfaces. He was one of the designers of Palladium and spent two years as an architect in the Tablet PC group. Currently, he is in Microsoft Research working on security, privacy, and fault-tolerance, especially for Internet-of-Things devices, and kibitzing in systems, networking, and other areas.

Leslie Lamport

Leslie Lamport's research has been centered on concurrency and fault-tolerance. He is the inventor of several well-known concurrent and distributed algorithms, including early algorithms for tolerating "Byzantine" faults. He did seminal work on the theory of cache coherence and distributed systems. He has also developed methods for formally specifying and verifying concurrent systems.

Lamport has received six honorary doctorates, the PODC Influential Paper Award, two Dijkstra Prizes in Distributed Computing, three ACM SIGOPS Hall of Fame Awards, two Jean-Claude Laprie Awards in Dependable Computing, the IEEE Piore Award in 2004, the IEEE John von Neumann Medal in 2008, and the ACM Turing award in 2014. He has been elected to the National Academy of Sciences, the National Academy of Engineering, and the American Academy of Arts and Sciences, and is a fellow of the Computer History Museum.

Christian Lengauer

Christian Lengauer received a Dipl.-Math. (1977) at the Free University in West-Berlin and an M.Sc. (1978) and a Ph.D. (1982) at the University of Toronto. He was an assistant professor at UT Austin (1982–1989), then a lecturer at the University of Edinburgh (1990–1992), and subsequently a full professor at the University of Passau (1992–2018). He is now retired.

His research activity in programming languages and methods focused on parallelization for high performance. He has been a major proponent of the polyhedron model for automatic loop parallelization. His research was generously supported by the national funding agencies in the USA, the UK and Germany, amounting close to 50 person-year altogether.

He was widely engaged in community building and support. He chaired the steering committee of the Euro-Par conference series for 17 years (2000–2017), was instrumental in the foundation of the IFIP WG 2.11 on program generation (2004), which he also chaired (2007–2013), and co-organized 10 Dagstuhl seminars on various topics. He has been holding a half-dozen mostly long-standing journal editorships.

Vladimir Lifschitz

Vladimir Lifschitz received a degree in mathematics from the Steklov Mathematical Institute (St. Petersburg, Russia) in 1971 and emigrated to the United States in 1976. Since 1990, he has been working at the University of Texas at Austin, where he is now Professor Emeritus.

He is a Fellow of the American Association for Artificial Intelligence and has served as the Editor-in-Chief of *ACM Transactions on Computational Logic*. He is a coinventor of answer set programming, a declarative programming methodology rooted in research on knowledge representation.

In 1987 and 1991, he received Publishers' Prizes for his contributions to International Joint Conferences on Artificial Intelligence. He received a Most Influential Paper in Twenty Years Award from the Association for Logic Programming in 2004, and a Test of Time Award in 2012.

Jayadev Misra

Jayadev Misra is the Schlumberger Centennial Chair Emeritus and University Distinguished Teaching Professor Emeritus at the University of Texas at Austin. He received his Ph.D. in Electrical Engineering from the Johns Hopkins University in 1972.

He works in the area of concurrent programming with emphasis on rigorous methods to improve the programming process. His work on the UNITY methodology, jointly with Mani Chandy, has been influential in both academia and industry. He and Mani Chandy (and, independently, Randy Bryant) pioneered the area of distributed discrete event simulation.

Misra is a member of the National Academy of Engineering and fellow of the ACM and IEEE. He has been awarded the Harry H. Goode Memorial Award of the IEEE in 2017 (jointly with Mani Chandy) and Doctor Honoris Causa by the École Normale Supérieure de Cachan, France, in 2010. He is a member of the Academy of Distinguished Teachers at the University of Texas at Austin and was awarded the Regents' Outstanding Teaching Award, University of Texas, in 2010. He is the author of two books, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988, coauthored with Mani Chandy, and *A Discipline of Multiprogramming*, Springer-Verlag, 2001.

Ernst-Rüdiger Olderog

Ernst-Rüdiger Olderog earned his Ph.D. at the University of Kiel in 1981. Since 1989, he is a professor of Theoretical Computer Science at the University of Oldenburg, Germany. In 1994, he was awarded the Leibniz Prize of the German Research Foundation (DFG) for his work on process theory.

His research interests include program verification, communicating processes, real-time systems, correct system design, and more recently games based on concepts of Petri nets. He has published three books on these subjects.

Brian Randell

Brian Randell graduated in Mathematics from Imperial College, London, in 1957 and joined the English Electric Company where he led a team that among others implemented the Whetstone KDF9 Algol compiler. Since 1969, he has been Professor of Computing Science at Newcastle University, where he is now Emeritus Professor and Senior Research Investigator.

He contributed to several areas in computer science, including compiler construction, operating systems, distributed secure systems, reliability, system dependability, and fault tolerance, and among others introduced the "recovery block" concept.

He has published over three hundred technical papers and reports and is coauthor or editor of seven books. He was the Chairman of the IEEE John von Neumann Medal Committee (2003-5) and the ACM A.M. Turing Award Committee (2005–2009). He has received a D.Sc. from the University of London, honorary doctorates from the University of Rennes and the Institut National Polytechnique of Toulouse, France, and the IEEE Emanuel R. Piore 2002 Award.

Hamilton Richards

Hamilton Richards graduated in mechanical engineering from Harvard in 1960. He worked for five years in aerospace and for two years as a Peace Corps Volunteer in Ethiopia. Returning to the US, he joined Fairchild Semiconductor, where he developed the operating system software for the experimental Symbol-2R computer. When the computer was relocated to Iowa State University, he followed.

After obtaining his Ph.D. at Iowa State in 1976, Richards joined Burroughs Corporation near San Diego. He moved to Austin in 1978 as a founding member of the company's Austin Research Center, working on various aspects of functional programming.

When the Austin laboratory closed in 1986, Richards joined The University of Texas at Austin as coordinator of the Year of Programming and Senior Lecturer in Computer Sciences. As the YoP wound down with the publication of four volumes of proceedings, he began lecturing full time, teaching courses in programming languages, functional programming, and formal methods.

Retired since 2006, Richards is devoting much of his time to the Citizens' Climate Lobby.

Fred B. Schneider

Fred B. Schneider is the Samuel B. Eckert Professor of Computer Science at Cornell University. Since 1996, he has also been a Professor-at-Large at the University of Tromso (Norway). Schneider's research concerns various aspects of trustworthy systems—systems that will perform as expected, despite failures and attacks.

The US National Academy of Engineering elected Schneider to membership in 2011, the Norges Tekniske Vitenskapsakademi (Norwegian Academy of Technological Sciences) named him a foreign member in 2010, and the American Academy of Arts and Sciences elected him to membership in 2017.

Schneider received the 2018 Edsger W. Dijkstra Prize in Distributed Computing and is also recipient of a Doctor of Science *honoris causa* from University of Newcastle, the IEEE Emanuel R. Piore Award, and the Jean-Claude Laprie Award in Dependable Computing.

Mikkel Thorup

Mikkel Thorup has a D.Phil. from Oxford University from 1993. From 1993 to 1998, he was at the University of Copenhagen. From 1998 to 2013, he was at AT&T Labs-Research. Since 2013, he has been back as Professor at the University of Copenhagen. He is currently a VILLUM Investigator heading Center for Basic Algorithms Research Copenhagen (BARC) supported by a € 5.3 million grant from the VILLUM Foundation.

Thorup is a Fellow of the ACM, a Fellow of AT&T, and a member of the Royal Danish Academy of Sciences and Letters. He is cowinner of the 2011 MAA Robbins Award in mathematics and winner of the 2015 Villum Kann Rasmussen Award for Technical and Scientific Research, which is Denmark's biggest individual prize for research. More recently, he was cowinner of the 2021 Fulkerson Prize and an ACM STOC 20-year Test of Time Award.

Mikkel's main work is in algorithms and data structures, where he has worked on both upper and lower bounds. Recently, one of his main focuses has been on hash functions unifying theory and practice. Mikkel prefers to seek his mathematical inspiration in nature, combining the quest with his hobbies of bird watching and mushroom picking.

Index

Abstract reduction systems, 172–173
Abstraction, 14, 146, 415
Acceptance test, 306
Action specifications, 73
Actions, 28
Active process, 72
AdaCore, 130
Address Resolution Protocol, 90–91
ALGOL 60, 10–11, 16, 298, 308, 377, 411–412, 494–495
ALGOL 60 Confidential, 468
Algol 68, 412–413, 437–438
Algol WG 2.1, 433–434
Algorithmic logic, 120
Algorithmic translators, 431
All-Pairs Shortest Paths (APSP), 25
Alternating path, 262
Alternative clauses, 316
AMD Optimizing CPU Libraries (AOCL), 225
Amdahl's law, 34
Angelic choice, 415
Angelic nondeterminism, 173, 196, 380
    expressive power of, 177–179
AOCL. *See* AMD Optimizing CPU Libraries (AOCL)
APIs. *See* Application Programming Interfaces (APIs)
Application Programming Interfaces (APIs), 221
APSP. *See* All-Pairs Shortest Paths (APSP)
Apt, Krzysztof R., 169, 373
arithmetic and geometric mean, 256–259
Association for Computing Machinery, 6
Asynchronous distributed system, 31
Asynchronous processes, 29
ATAC. *See* Austin Tuesday Afternoon Club (ATAC)
Atomic operations, 343
Atomic queries, 176
Atomic shared memory registers, 55
Atomicity, 31–32, 95
Austin Tuesday Afternoon Club (ATAC), 205, 402, 459, 464, 466, 476, 480
Automated model, 109–110
Automatic computers, 17
Automatic program synthesis, 400
Automatic theorem proving, 176
Automation, 124
Axiomatic semantics, 111, 419

Backtrack nondeterminism, 195
Backus-Naur form (BNF), 10–11, 15, 387
Backward error analysis, 224

Banker’s algorithm, 378*n*6
BARC. *See* Burroughs Austin Research Center (BARC)
Batching, 34
Bijlsma, Lex, 455–456
Binary semaphore, 54
60th birthday salute, 441–442
Bisimilarity, 196
Bisimulation, 196
Blob, 34
Block number, 298
BNF. *See* Backus-Naur form (BNF)
Boolean expressions, 182
Boolean functions, 334
Bootstrapping, 427
“Bottom–up” parsing technique, 427
Bounded buffer program, 148
British Computer Society, 6
Broy, Manfred, 493
Burroughs Austin Research Center (BARC), 482–483
Burroughs Compiler Game, 428
Byzantine processes, 97

CACM. *See Communications of the ACM* (CACM)
Calculational proofs, 391
- natural deduction *vs*., 235–238
- predicate transformers, 241–243
- propositional calculations, 238–240
- semi-formal, 233–234
- smooth reasoning, 234–235

“Call by name” parameter mechanism, 494
“Call by value” parameter mechanism, 494
Calvert, Ken, 456–459
CBC. *See* Correctness-by-construction (CBC)
Chandy, K. Mani, 459–461
Choice clauses, 316
Choice nondeterminism, 195
Chomsky hierarchy of formal grammars, 177
Church–Rosser property, 172
Closure, 89
Coarse-grained algorithm, 59–61
COBOL, 389
Code generation, 17
Code inspection, 119
Collector, 65–66, 341, 344, 347
Combining function, 34
Commands, 28
Commit point, 37
Communicating sequential processes (CSP), 186–190, 413
Communications of the ACM (CACM), 47, 64
Commuting, 31–32
Comparison-based sorting lower bound, 23
Compassion, 193
Composing programs, 15
Compositionality, 89
Computability, 169–170
Computably ambiguous functions, 174
Computation tree, 191, 195, 399
Concurrency, 28
- easy, 36–42
- hard, 42–43
- language constructs for, 29
- taxonomy of, 32–33

Concurrent algorithms, 47
- mutual exclusion, 47–58

on-the-fly garbage collection, 64–71
self-stabilization, 58–64
termination detection, 71–78
Concurrent programming control, 311
problem, 311–312
proof, 313
solution, 312–313
Conditional clauses, 316
Conditional expressions, 182
Conference on Software Engineering in Garmisch, 12, 437, 501, 506
Configurational abstraction, 365
Consensus model, 109
Consistency, 33
Context-free grammar, 171
Context-free languages, 178
Convergence, 88–89
Cooperating sequential processes, 324, 378
Cooperation of concurrent processes, 340
fine-grained solution, 354–358
first coarse-grained solution, 345–352
grain of action, 342–343
new coarse-grained solution, 352–354
Correctness in presence of round-off error, 224
Correctness-by-construction (CBC), 106–107, 130–131
Coxeter's rabbit, 262–263
Critical section, 36, 48, 311–312
CSP. *See* Communicating sequential processes (CSP)

Deadlock, 38
freedom, 50–52
Deadly embrace, 38
Debugging, 361
Declaration entries, 427
Deductive reasoning, 155
Dekker's algorithm, 51–53
Demonic choice, 415
Demonic nondeterminism, 180, 196
Dense linear algebra software (DLA software), 216, 224–225
Deriving algorithms, 218–221
Design experience, 325–327
Design fault tolerance, 437
Deterministic automaton, 178
DEUCE, 424, 427–428
Dial's algorithm, 24
Diffusing computation, 91
Dijkstra algorithm, 21–23, 53–54, 392
"Dijkstra for the Masses", 158
Dijkstra's lemma, 248–249
Dijkstra, Edsger Wybe, 5, 47, 205, 207–212, 247, 287, 291, 301, 311, 315, 319, 333, 339, 359
Archive, 511–513
biography, 374–375
character and lifestyle, 390–392
decades of research, development, and impact, 221–225
early contributions, 377–381
fairness, 400–402
Fermat's little theorem, 252–254
nineties, 385–386
opinions and attitudes, 386–390
Dijkstra–Prim minimum spanning tree algorithm, 250–251
Dijkstra–Scholten overloading convention, 241–242
Dining philosophers, 57–58
Display, 429–430

Distance convergence on circle, 87–89
Distributed control, 334
Distributed systems, 28, 30, 93
Divide and conquer, 144–145
DLA software. *See* Dense linear algebra software (DLA software)
Dynamic index, 317
Dynamic logic, 120–123
Dynamo, 44

EASICODE, 424
Easy concurrency, 36
 monitors, 40–41
 scheduling, 41–42
EC. *See* Eventual consistency (EC)
EDSAC, 9
Elliott computers, 411
Emerson, E. Allen, 399
Enumeration, 145
Enumerative reasoning, 366
Epochs, 37, 39
Erratic nondeterminism, 196
Error checking scheme, 426
Event graphs, 335
Eventual consistency (EC), 43–44, 90
Everywhere operator, 160–161
EWD1171 critique, 258–259
EWD1273 critique, 251–252
EWD1318 critique, 263
EWD742 critique, 255–256
External nondeterminism or alternation, 198

Failure equivalence, 199
Fair scheduling, 94
Fairness, 190–194, 400–402, 472–473
Fault containment, 97
Fault-tolerance, 96–97
Features, 211
 orientation, 211
Fermat's little theorem, 252–254
Fibonacci heaps, 23
FLAME methodology. *See* Formal Linear Algebra Methods Environment methodology (FLAME methodology)
FLAME notation, 210, 222–223
Flowchart programs, 111
Floyd's approach to nondeterministic programming, 174–176
"Fog Index" measure of comprehensibility, 430
Formal Linear Algebra Methods Environment methodology (FLAME methodology), 216–217
Formal mathematics in programming, 501–504
Formal model, 109–110
FORTRAN, 10–11, 368
Forward reasoning, 128–129
Functional programming, 388
Fusion, 35

Garbage collection, 65, 340
Garbage collector, 341
Garbage nodes, 340
Ghost variables, 448
go to statement, 315–318, 364
Grammars, 170–171
Graph-theoretic algorithms, 96
Gries, David, 141, 405
Guard, 381
Guarded commands, 112–114, 157, 179–182, 347, 414

Hähnle, Reiner, 105

Hall condition (HC), 260
Hall's theorem on distinct representatives, 259–262
Hard concurrency, 42–43
Haskell-style equational reasoning, 209
HASP System. *See* Houston Automatic Spooling Priority System (HASP System)
HC. *See* Hall condition (HC)
Hehner, Eric C. R., 461–464
Herman, Ted, 81
Hesselink, Wim H., 464–466
Hierarchical systems, 17–18
Hint calculus, 456
Hoare logic, 28, 121
Hoare triple, 111, 155
Hoare, Tony, 411
Houston Automatic Spooling Priority System (HASP System), 438

IBM, 433–434
IBM and Algol WG 2.1, 433–434
Idle process, 72
Indeterminism, 380
Inductive invariant, 42, 69, 71
Informal model, 109–110
Integer priority queues, 24
Integer weights, 24–25
Intellectually manageable programs, 13–14, 362
Interference freedom, 446
Internal nondeterminism, 198
Interrupt routines, 27–28

Java Card, 130
Joshi, Rajeev, 466–468
Jump instruction, 318
Justice, 192

K-state machine solution, 335
KDF9 Algol programs, 425, 427–428
Kernel calls, 27–28
Knowledge, 223
Knuth, Don, 468–470
Knuth–Morris–Pratt algorithm, 470
Kruskal's algorithm, 248
Kruskal's algorithm for minimum spanning tree, 248–252

Lamport, Leslie, 47, 339
Lampson, Butler, 27
Language constructs for concurrency, 29
LAPACK, 221
LaTeX, 515–516
Left commutes, 31
Left mover, 31
Legitimate state, 83
Leibniz's substitution of equals for equals, 158–161
Lengauer, Christian, 205
Libflame, 225
Lifschitz, Vladimir, 231
Linear Temporal Logic (LTL), 401
Linearizability, 33
LINPACK, 220–221
LISP, 10
LLVM, 210*n*4
Local variable, 342
Lock, 39
Lock invariant, 37–38
Logic programming, 176–177
Logic programming and Prolog, 176–177
Loose nondeterminism, 195
LTL. *See* Linear Temporal Logic (LTL)

Map-reduce, 36

Marktoberdorf Summer School, 493–494
 arena, 498–499
 beginnings, 496–497
 Edsger as director of, 497–498
 Edsger's views, 504–505
Martin, Alain J., 339, 470–472
Massive Open Online Course (MOOC), 226
Mathematical centre, 424–425
Mathematical induction, 146, 367
McCarthy's ambiguity operator, 173–174
Mechanized model, 109–110
Mental aids, 145–147
Message interpreter, 324
Microprocessors, 414
Minimum spanning tree, 248
Misra, Jayadev, 247, 472–474
Modal logic, 122
Modal operators, 242
Model checking, 400
Model exploration, 96
Modeling parallel programs, 184–186
Modern multiprocessors, 32
Monitors, 40–41
MOOC. *See* Massive Open Online Course (MOOC)
Mounting, 35
Multi-version concurrency control (MVCC), 43
Multicore processors, 28
Multiprogramming system, 321
Mutator, 65–66, 68, 341, 344
Mutual exclusion, 27, 36, 47, 84, 333
 Dekker's algorithm, 51–53
 Dijkstra's algorithm, 53–54
 dining philosophers, 57–58
 problem, 55–57
 semaphores, 54–55
Mutual exclusion problem, 378*n*5
MVCC. *See* Multi-version concurrency control (MVCC)
Myers, Maggie, 215

National Research Development Corporation (NRDC), 441
1968 NATO Conference on Software Engineering, 150–153, 438
1968 NATO S/W Engineering Conference, 437
Natural deduction, 158–159, 235–238
Naumann, David A., 475–478
Nested equivalences, 233
Non-interference principle, 145, 162
Noncritical section, 48
Nondeterminacy, 380
Nondeterminism, 27, 169, 380
 angelic nondeterminism, 173–179
 avoiding, 170–173
 in context, 196–199
 demonic nondeterminism, 180, 196
 taxonomy, 195–196
Nondeterministic automaton, 178
Nondeterministic finite automata, 173
nondeterministic finite automata and Turing machines, 173
Nondeterministic scheduling, 82
NRDC. *See* National Research Development Corporation (NRDC)
Nuisance concurrency, 44
Null pointers, 65
Numerical information, 293

Object, 34
Object-oriented programming, 389–390

OCC. *See* Optimistic concurrency control (OCC)
OCP. *See* On concurrent programming (OCP)
Olderog, Ernst-Rüdiger, 169
On concurrent programming (OCP), 446–450
On-the-fly garbage collection, 64
  algorithm and significance, 70–71
  problem, 64–66
  solution, 66–68
  verification, 68–70
One-bit algorithm, 49
One-bit protocol, 48, 53
Operational abstraction, 146, 150, 364, 367
Operational reasoning, 445
Operational semantics, 110, 419
Optimistic concurrency control (OCC), 39
Orderly program development, 148

Parallel algorithm, 47
Parallel computing, 222
Parallel programs, 368
Parameter pointer, 297
Partial failures, 30
Partitioned matrix expression (PME), 217
Partitioning. *See* Sharding
PASCAL, 368
Path condition, 118
Phase ordering problem, 224
PHC. *See* Post hoc verification (PHC)
Pipelining. *See* Streaming
PME. *See* Partitioned matrix expression (PME)
PODC. *See Principles of Distributed Computing* (PODC)
Polyhedron model, 210
Post hoc verification (PHC), 130–131
Post hoc verification (PHV). *See* Posteriori verification
Posteriori verification, 106–107
Power set construction, 173
Predicate algebra, 161
Predicate calculus, 242
Predicate transformer, 114, 241–243
Principles of Distributed Computing (PODC), 64
Priority queues, 23
Privileges, 334
  circulation, 84
Probe, 76–77
Process fairness, 56
Processor allocation, 323–324
Producer–consumer synchronization, 27
Program correctness, 141–142
Program family, 211
Program refinement, 211
Program sequencing, 316
Program testing, 105
Program verification, 105, 109, 327
  Dijkstra’s view, 106–110
  discipline, 131–132
  theory and practice, 130
  weakest precondition calculus and context, 110–123
Programming concurrent systems, 27–28
  atomicity and commuting, 31–32
  concurrency, 28–31
  easy concurrency, 36–42
  eventual consistency, 43–44
  hard concurrency, 42–43
  nuisance concurrency, 44
  sharding, 33–35

streaming, 35–36
taxonomy of concurrency, 32–33
Prolog, 176–177
Propositional calculations, 238–240
Pythagorean theorem, 382

Query, 176–177

Randell, Brian, 423
Rao, J. R., 478–482
Readers/writers problem, 57
Recursion, 377
Recursive programming, 291
aim, 291
consequences, 295
link, 295–297
recursive techniques and ALGOL 60, 298
stack, 291–293
stacked reservations, 293–295
Referential transparency, 208
Refinement, 365
Relational calculus, 119–120
Relational semantics, 121
"Relaxed consistency" multiprocessor memory, 44
Reliability program, 360–361
Repetition clauses, 316
Representational abstraction, 146, 150, 365
Representational refinement, 365
"Reverse Polish" notation, 427
Richards, Hamilton, 482, 511
Right commutes, 31
Right mover, 31
Rigorous model, 109–110
Ring algorithm, 73–77
Robustness, 334
Roots, 340

Sampling, 34
Scalable commutativity rule, 32
ScaLAPACK, 221
Schedulers, 95
Scheduling, 41–42
Scheevel, Mark, 484–486
SCHEMER language, 174
Schneider, Fred B., 445
Scholten, C.S., 339
Science of programming, 393
SDR. *See* System of distinct representatives (SDR)
Secretaries, 41
Secretary, 413. *See also* Secretaries
Segment controller, 324
Segment variable, 323
Self-stabilization, 30, 58, 81, 334, 379–380
adoption, 91–92
coarse-grained algorithm, 59–61
complexity and verification, 89
distance convergence on circle, 87–89
finer-grained algorithm, 62
impact, 92–97
paper's influence, 63–64
privileges in array, 85–87
privileges in ring, 82–85
problem, 58–59
relevance, 90–91
Self-stabilizing systems, 334, 438
four-state machine solution, 336
K-state machine solution, 335
three-state machine solution, 336–337
Semantic relations, 208
Semantics-preserving program transformation, 207
Semaphores, 36, 54–55, 378*n*5

Sequencing discipline, 364
Sequential processes, 27, 29, 325
Sequential systems, 28
Serializability, 33
Serialized processes, 37
Sharding, 33–35
Shards, 33
Sharing, 334
Shortest Path Algorithm, 21
Silent stabilization, 93
Single source shortest path problem (SSSP), 21
Snap-stabilization, 97
Soft errors, 90
Software crisis, 393, 437, 493
Software engineering, 151
Software failure, 359
Software product line, 211
SOSP. *See* Symposium on Operating System Principles (SOSP)
Spin lock, 41
SSSP. *See* Single source shortest path problem (SSSP)
Stack, 291–293, 429
Stacked reservations, 293–295
Staircase method, 89
Starvation, 84*n*2
Starvation freedom, 52
Static Chain, 429
Steam method, 326
Steffens, E.F.M., 339
Stepwise program construction, 142
Storage allocation, 322–323
Straight-line programs, 116
Streaming, 35–36
Streamlining mathematical arguments, 247
Striping. *See* Sharding
Strong fairness, 193, 401
Structured programming, 147–150, 247, 379
Subroutine library, 9
Symbolic execution, 117–119
Symposium on Operating System Principles (SOSP), 434–435
Synchronization, 333
Syntax, 412–413
System hierarchy, 324–325
System of distinct representatives (SDR), 259
System structure, 322–325
Systolic algorithms, 439
Systolic array, 210

Taxonomy of concurrency, 32–33
Taxonomy of nondeterminism, 195–196
Technische Hogeschool Eindhoven (THE), 378, 453
Termination detection, 71, 91
  history, 77–78
  problem, 72
  ring algorithm, 73–77
  tree algorithm, 72–73
Testing processes, 197
THE. *See* Technische Hogeschool Eindhoven (THE)
THE multiprogramming system, 151, 319–331, 378
THE Operating System, 435–436
Third generation computers, 10
Thorup, Mikkel, 21
Thread, 30
TOMS. *See* Transactions on Mathematical Software (TOMS)
“Top–down recursive descent” technique, 427

Transactions, 32
processing systems, 32
Transactions on Mathematical Software (TOMS), 223
Transient faults, 90
Translation, 519
Translator, 308
stack, 427
Tree algorithm, 72–73
Trivial implementation, 23
Turing machines, 169, 173
Turner, David, 486–489

Unary predicate, 241
Unblocked algorithm, 218
Unbounded nondeterminism, 191, 195
Unconditional fairness, 402
University of Texas (UT), 456
Use entries, 427–428
UT. *See* University of Texas (UT)

van de Geijn, Robert, 215
VCG architecture. *See* Verification condition generator architecture (VCG architecture)
Verification, 89, 105–106
Verification condition generator architecture (VCG architecture), 127–128
Virtual machine, 30

Wait-free algorithm, 71
Wait-free concurrency, 43
Weak fairness, 192
Weakest liberal precondition, 114
Weakest precondition calculus (wp-calculus), 110, 114–117
symbolic execution of, 117–119
Weakest preconditions, 154–158, 399–400
reasoning, 129
semantics, 179, 381
Whetstone Algol Compiler, 425
While-programs, 111
Williams' heap, 23
Wilson's theorem, 254–255
WORDS. *See* Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)
Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), 423
Wp-calculus. *See* Weakest precondition calculus (wp-calculus)

Edsger Wybe Dijkstra
His Life, Work, and Legacy

Krzysztof R. Apt, Tony Hoare (Editors)

Edsger Wybe Dijkstra (1930–2002) was one of the most influential researchers in the history of computer science, making fundamental contributions to both the theory and practice of computing. Early in his career, he proposed the single-source shortest path algorithm, now commonly referred to as Dijkstra's algorithm. He wrote (with Jaap Zonneveld) the first ALGOL 60 compiler, and designed and implemented with his colleagues the influential THE operating system. Dijkstra invented the field of concurrent algorithms, with concepts such as mutual exclusion, deadlock detection, and synchronization. A prolific writer and forceful proponent of the concept of structured programming, he convincingly argued against the use of the Go To statement. In 1972 he was awarded the ACM Turing Award for "fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design." Subsequently he invented the concept of self-stabilization relevant to fault-tolerant computing. He also devised an elegant language for nondeterministic programming and its weakest precondition semantics, featured in his influential 1976 book *A Discipline of Programming* in which he advocated the development of programs in concert with their correctness proofs. In the later stages of his life, he devoted much attention to the development and presentation of mathematical proofs, providing further support to his long-held view that the programming process should be viewed as a mathematical activity.

In this unique new book, 31 computer scientists, including five recipients of the Turing Award, present and discuss Dijkstra's numerous contributions to computing science and assess their impact. Several authors knew Dijkstra as a friend, teacher, lecturer, or colleague. Their biographical essays and tributes provide a fascinating multi-author picture of Dijkstra, from the early days of his career up to the end of his life

ABOUT ACM BOOKS

ACM Books is a series of high-quality books published by ACM for the computer science community. ACM Books publications are widely distributed in print and digital formats by major booksellers and are available to libraries and library consortia. Individual ACM members may access ACM Books publications via separate annual subscription.

ISBN 978-1-4503-9771-1
90000
9 781450 397711

BOOKS.ACM.ORG · WWW.MORGANCLAYPOOLPUBLISHERS.COM