Computation and State Machines

Leslie Lamport

19 April 2008

Preface

For quite a while, I've been disturbed by the emphasis on language in computer science. One result of that emphasis is programmers who are C++ experts but can't write programs that do what they're supposed to. The typical computer science response is that programmers need to use the right programming/specification/development language instead of/in addition to C++. The typical industrial response is to provide the programmer with better debugging tools, on the theory that we can obtain good programs by putting a monkey at a keyboard and automatically finding the errors in its code.

I believe that the best way to get better programs is to teach programmers how to think better. Thinking is not the ability to manipulate language; it's the ability to manipulate concepts. Computer science should be about concepts, not languages. But how does one teach concepts without getting distracted by the language in which those concepts are expressed? My answer is to use the same language as every other branch of science and engineering—namely, mathematics. But how should that be done in practice? This note represents a small step towards an answer. It doesn't discuss how to teach computer science; it simply addresses the preliminary question of what is computation.

Contents

1 Introduction

Much of computer science is about state machines. This is as obvious a remark as saying that much of physics is about equations. Why state something so obvious?

Imagine a world in which physicists did not have a single concept of equations or a standard notation for writing them. Suppose that physicists studying relativity wrote the "einsteinian" $m \nearrow c^2 \hookleftarrow E$ instead of $E = mc^2$, while those studying quantum mechanics wrote the "heisenbergian" $\underset{\wedge}{E} \quad \overset{c^2}{m}$; and that physicists were so focused on the syntax that few realized that these were two ways of writing the same thing. In such a world, it would be worth observing that relativity and quantum mechanics both used equations.

This imagined world of physics seems absurd. Its analog is the reality of computer science today. Computation is a major topic of computer science, and almost every object that computes is naturally viewed as a state machine. Yet computer scientists are so focused on the languages used to describe computation that they are largely unaware that those languages are all describing state machines.

Teaching our imaginary physicists that einsteinians and heisenbergians are different ways of writing equations would not lead to any new physics. The equations of relativity are different from those of quantum mechanics. Similarly, realizing that so much of computer science is about state machines might not change the daily life of a computer scientist. The state machines that arise in different fields of computer science differ in important ways, and they may be best described with different languages. Still, it seems worthwhile to point out what they have in common.

State machines provide a framework for much of computer science. They can be described and manipulated with ordinary, everyday mathematics—that is, with sets, functions, and simple logic. State machines therefore provide a uniform way to describe computation with simple mathematics.

The obsession with language is a strong obstacle to any attempt at unifying different parts of computer science. When one thinks only in terms of language, linguistic differences obscure fundamental similarities. Simple ideas can become complicated when they must be expressed in a particular language. A recurring theme is the difficulty that arises when necessary concepts cannot be introduced either because the language has no way of expressing them or because they are considered to be politically incorrect. (A number of different terms have been used to mean politically correct, such as "fully abstract", "observable", and "denotational".)

The purpose of this note is to indicate how computation is expressed with state machines and how doing so can clarify the concepts underlying the disparate languages for describing computations. Section 2 explains what state machines are and how they can describe things that compute, and Section 3 considers the important example of computer programs. Section 4 shows how state machines are described with mathematics. Section 5 delves more deeply into the use of state machines in the area of computer science that can be called *correctness*.

Many of the ideas expressed here appear, sometimes implicitly, in the work of Yuri Gurevich and others on Abstract State Machines [7, 13, 15]. Much of that work has been motivated by the desire to describe the class of effectively executable computations [14]. Readers who have studied Abstract State Machines will find much here that is familiar, but viewed from a somewhat different perspective.

2 State Machines

2.1 Behaviors

Much of computer science is about *computing objects*—objects that compute. I begin by considering what a computation is. There are several ways to define *computation*. For now, I take the simplest: a computation is a sequence of steps, which I call a *behavior*. There are three common choices for what a step is, leading to three different kinds of behavior:

Action Behavior A step is an *action*, which is just an element of some set of actions. An action behavior is a sequence of actions.

State Behavior A step is a pair $\langle s, t \rangle$ of states, where a state is an element of some set of states. A state behavior is a sequence $s_1 \to s_2 \to s_3 \to \cdots$ of states. The step $\langle s_i, s_{i+1} \rangle$ represents a transition from state s_i to state s_{i+1}.

State-Action Behavior A step is a triple $\langle s, \alpha, t \rangle$, where s and t are states and α is an action. A state-action behavior is a sequence $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \xrightarrow{\alpha_3} \cdots$. The step $\langle s_i, \alpha_i, s_{i+1} \rangle$ represents a transition from state s_i to state s_{i+1} that is performed by action α_i.

Behaviors can be finite or infinite. A finite behavior is said either to *terminate* or to *deadlock*, depending on whether or not we like its final state (or its final action, for action behaviors). When behaviors are expected to be

finite, sometimes infinite behaviors are said to *diverge* and finite behaviors that end in undesired states are said to *abort*.

State and state-action behaviors are essentially equivalent. A state behavior can be regarded as a state-action behavior with only a single dummy action $\bot$. A state-action behavior can be represented as a state behavior whose states are pairs, where $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \xrightarrow{\alpha_3} \cdots$ is most conveniently represented as $\langle \bot, s_1 \rangle \to \langle \alpha_1, s_2 \rangle \to \langle \alpha_2, s_3 \rangle \to \cdots$. Action behaviors are usually specified by defining state-action behaviors and throwing away the states.

2.2 State Machines

A computing object generates computations. When a computation is a state behavior, such an object is naturally defined as a state machine. A state machine is usually specified by a set $\mathcal{S}$ of states, a set $\mathcal{I}$ of initial states, and a next-state relation $\mathcal{N}$ on $\mathcal{S}$, so $\mathcal{I} \subseteq \mathcal{S}$ and $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$. It generates all state behaviors $s_1 \to s_2 \to s_3 \to \cdots$ such that:

S1. $s_1 \in \mathcal{I}$

S2. $\langle s_i, s_{i+1} \rangle \in \mathcal{N}$, for all i.

S3. If the behavior is finite, then it ends in a state s for which there is no state t with $\langle s, t \rangle \in \mathcal{N}$.

A state-machine is said to be *deterministic* iff the next-state relation $\mathcal{N}$ is a function—that is, iff for each state s there is at most one state t with $\langle s, t \rangle \in \mathcal{N}$.

Nondeterministic state machines may also include fairness conditions that require certain steps to occur if they are possible. In this case, S3 can be a fairness condition that may or may not be required.

If a computation is a state-action behavior, then a computing object is specified with a state-action machine. Such a machine is like a state machine except that the next-state relation becomes a next-state set $\mathcal{N}$ that is a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$, where $\mathcal{A}$ is the set of all actions. A state-action machine is deterministic iff for every state s there is at most one pair $\langle \alpha, t \rangle$ with $\langle s, \alpha, t \rangle \in \mathcal{N}$.

Here are a few examples of computing objects and how they are described by state or state-action machines.

Automata Many kinds of automata have been defined, the most well-known being the Turing machine. A Turing machine is a state machine

whose state describes the contents of the tape, the internal state, and the position of the read/write head. We can let the set of initial states describe all tapes that represent valid inputs, or we can let each possible input tape define a different state machine, so the initial-state set $\mathcal{I}$ contains only a single initial state. It should be obvious how other kinds of automata are also naturally described as state or state-action machines—for example, Moore machines, Mealy machines, and other finite-state automata; pushdown automata; multi-tape Turing machines; and cellular automata.

von Neumann Computers The state of a von Neumann computer specifies the contents of the memory and of all the registers, including a program counter (*pc*) that contains the address of the next instruction to be executed. The next-state relation contains the pair $\langle s, t \rangle$ of states iff executing the next instruction (the one specified by *pc*) in state s produces state t. Output can be represented with an output register; input can be represented with a *read* instruction that nondeterministically sets a memory location to an arbitrary value. (There are a number of other ways to represent input and output as well.) We usually let the set $\mathcal{I}$ of initial states contain all possible states. However, if we are interested in a particular program executed on the computer, we can let $\mathcal{I}$ be the set of all states containing the program in some portion of memory, having legal inputs in the appropriate memory locations, and with *pc* containing the address of the program's first instruction.

Algorithms An algorithm is usually considered to be a recipe for generating behaviors. For a sequential algorithm, computational complexity measures how many steps are in the behaviors it generates.[1] As an example of a concurrent algorithm, consider a distributed message passing algorithm in which a set P of processes interact by sending and receiving messages. A state is the cross product of local states of all processes together with the state of the communication medium, which describes the messages currently in transit. For a dynamic system in which processes enter and leave, a process that is not currently part of the system has a default *inactive* state. (If there is no bound on the number of processes that can become active, then the set P is

[1] Some computer scientists think an algorithm is a function from inputs to outputs. If that were true, then bubble sort and heap sort would be the same algorithm, since they compute the same function.

infinite.) The next-state relation $\mathcal{N}$ is the union of:

- For each process p in P, a relation $\mathcal{N}_p$ that describes the steps performed by p. In a typical step, p receives a message and responds by sending zero or more messages. If $\langle s, t \rangle$ is in $\mathcal{N}_p$, then s and t can differ only in the local state of p and the state of the communication medium.
- A relation $\mathcal{C}$ that describes internal steps of the communication medium—for example, the loss of one or more messages. If $\langle s, t \rangle$ is in $\mathcal{C}$, then s and t can differ only in the state of the communication medium.

It is also possible to define $\mathcal{N}$ so it contains steps that are performed simultaneously by multiple processes or by one or more process and the communication medium. The use of state machines that allow such simultaneous operations is sometimes (rather misleadingly) called "true concurrency".

BNF Grammars A BNF grammar can be described as a state machine whose states are sequences of terminals and/or non-terminals. The set of initial states contains only the sequence consisting of the single starting non-terminal. The next-state relation is defined to contain $\langle s, t \rangle$ iff s can be transformed to t by applying a production rule to expand a single non-terminal.

Process Algebras A process algebra such as CCS [21] can be described by state-action machines whose states are terms of the algebra. The next-state set $\mathcal{N}$ contains $\langle s, \alpha, t \rangle$ iff $s \xrightarrow{\alpha} t$ is a transition of the algebra. A term defines the state-action machine in which the set of initial states contains only that term.

It would be an absurd trivialization to say that Turing machines and distributed algorithms are the same because they are state machines, just as it would be absurd to say that relativity and quantum mechanics are the same because they use equations. However, we would be suspicious if the mathematics used to reason about equations depended on whether they were written as einsteinians or as heisenbergians. Likewise, we should be suspicious if completely different formalisms are used to prove termination of Turing machines and of distributed algorithms. On the other hand, there is a big difference between finite and infinite sets of states, so we would not be surprised if the methods used to prove termination of finite state automata

and von Neumann computers were different from those used for Turing machines and distributed algorithms. (In practice, the number of states of a von Neumann computer is so large that one proves its termination by the same methods used for infinite-state state machines.)

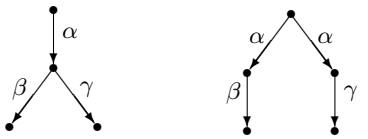
2.3 Other Kinds of Computations

A state machine is a generator of computations. So far, I have taken a computation to be a behavior. Other definitions of *computation* have been proposed. I now briefly describe how a state or state-action machine is considered to generate them.

One alternate definition of a computation is a state tree, which is a tree whose nodes are labeled by states. A state tree describes the tree of possible executions from the root's state. A state machine generates every state tree for which (i) the root node is in the set of initial states and (ii) a node labeled with any state s has a child labeled with state t iff $\langle s, t \rangle$ is in the next-state relation.

We can also define a computation to be a state-action tree, which is a state tree whose edges are labeled by actions. A state-action machine can be viewed in the obvious way as a generator of state-action trees, where there is an edge labeled α from a node labeled with state s to a child labeled with state t iff $\langle s, \alpha, t \rangle$ is in the next-state set.

As with state-action behaviors, some formalisms delete the state labels from state-action trees to obtain action trees with only the edges labeled. This leads some to argue that we must describe computations with trees because behaviors are not expressive enough. The classic example asserts that the two trees



are generated by different machines that both generate the two action behaviors

$$\cdot \xrightarrow{\alpha} \cdot \xrightarrow{\beta} \cdot \qquad \text{and} \qquad \cdot \xrightarrow{\alpha} \cdot \xrightarrow{\gamma} \cdot$$

Hence, it is claimed, two different systems generate the same behaviors. In fact, the two systems generate the same behaviors only if political correctness prevents mentioning the states, so one views only the actions. The state-action behaviors generated by the state-action machines that produce the

two action trees are different because the machines have different sets of states.

Another definition of a computation is a partially ordered multiset of actions, called a *pomset* for short. A pomset is a set Π with an irreflexive partial-order relation $\prec$, where the elements of Π are labeled with actions. An element labeled by action α represents an execution of α, and $e \prec f$ means that the action execution represented by e precedes the action execution represented by f. If neither $e \prec f$ nor $f \prec e$ holds, then the two executions are said to be concurrent. The pomsets generated by a state-action machine are defined to consist of all pomsets $\pi(\sigma)$ such that σ is a state-action behavior generated by the machine, where $\pi(\sigma)$ is defined intuitively as follows. Dropping the states from σ yields an action behavior (a sequence of actions) that we can view as a totally ordered multiset of actions. (It is a multiset because the same action can appear multiple times in the computation.) We define $\pi(\sigma)$ to be the pomset obtained from this totally ordered multiset by eliminating orderings between pairs of elements that represent concurrent action executions. Readers interested in pomsets will find a precise definition of $\pi(\sigma)$ in the appendix.

For deterministic state machines or state-action machines, the trees they generate consist of a single path and the pomsets they generate are totally ordered. Hence their trees and pomsets are equivalent to behaviors. The different kinds of computations differ only for nondeterministic machines.

Different kinds of computation have different uses. Behaviors are natural for discussing termination of a distributed algorithm, which means that all of its behaviors are finite. Trees are natural for discussing termination of a nondeterministic Turing machine, which usually means that its state tree contains at least one leaf node.[2] It can be difficult to define what fairness conditions mean when computations are trees, so computations are most often taken to be behaviors when fairness is important.

2.4 Other Ways to Describe Computations

Methods for describing computations by writing state machines have been criticized for introducing extraneous details. Mentioning certain parts of the state has been considered politically incorrect, and methods have been proposed that avoid mentioning them—or even mentioning the state at all.

There is sometimes good reason why some part of the state should not be described by a computation. For example, a specification of a memory

[2] A nondeterministic Turing machine is usually defined to terminate iff any of its possible executions does.

should describe the sequences of reads and writes that are permitted. The contents of the memory can be considered part of the implementation and should not be mentioned. Thus, we might describe a memory by a set of action behaviors, where an action is something like *set location 14 to* -7 or *read 33 from location 14*. Alternatively, we might describe it as a set of state behaviors, where the state describes only the contents of the memory's input and output registers, not the contents of memory locations.

We can use state machines to write such specifications by simply declaring the unwanted part of the state to be *hidden* and not considered to be part of a computation [1]. Methods in which a computation is taken to be an action behavior often use state-action machines in which the entire state is considered hidden—for example, I/O Automata [19].

Some computer scientists believe we should not mention the unwanted part of the state, even if it is hidden. An alternative is to describe computations by a set of axioms—for example, using temporal logic [25]. However, this simply does not work in practice for any but the most trivial examples. The only practical method for describing nontrivial sets of computations is with state machines. Methods that apparently describe computations directly without using hidden state work only if they can essentially encode a state machine. For example, one can specify a set of computations with a relation R between input streams and output streams, where R is defined recursively using auxiliary functions that describe the hidden state [8].

3 Programs as State Machines

Programs generate computations, so they are obvious candidates to be viewed as state machines. However, most computer scientists seem to think about programs primarily in terms of their syntax, not their computations. Consider the three C programs of Figure 1 that compute 7!. When asked which of them differs the most from the other two, computer scientists usually answer Program 3. The reason they give is that Programs 1 and 2 use iteration while 3 uses recursion, or perhaps that 1 and 2 are imperative while 3 is functional. However, iteration and recursion are different ways of expressing computations; they do not necessarily express different computations. In terms of their computations, it is Program 2 that differs most from the other two. The significant steps in computing 7! are the multiplications. Programs 1 and 3 perform the same sequence of multiplications, which is different from the sequence performed by Program 2. All three produce the same result only because multiplication is commutative. (To see this, try

Program 1:
```
#include <stdio.h>
main() { int f = 1, i = 2;
         for (i = 1; i <= 7; ++i) f = i * f;
         printf ("%d", f) ;
       }
```

Program 2:
```
#include <stdio.h>
main() { int f = 1, i ;
         for (i = 7; 1 < i; --i) f = i * f;
         printf ("%d", f) ;
       }
```

Program 3:
```
#include <stdio.h>
int fact(int i)
      { return (i == 1) ? 1 : i * fact(i-1); }
main() { printf ("%d", fact(7)); }
```

Figure 1: Three C programs for computing 7!.

replacing "`*`" with "`-`" in the programs and running them.)

To describe a program as a state machine, we must decide what constitutes a step. How many steps are performed in executing the statement $f = i * f$ of the programs in Figure 1? Does a single step change the value of f to the product of the current values of i and f? Are the evaluation of $i * f$ and the assignment to f two steps? Are there three steps—the reading of i, the reading of f, and the assignment to f?

The output produced by executing a sequential program does not depend on how many steps are performed in executing a statement like $f = i * f$. We can make a fairly arbitrary decision of what constitutes a step when describing a sequential program as a state machine. For a C program, the state will describe what program variables are currently defined, their values, the contents of the heap and of the program's call stack, the current control point, and so on. Specifying how to translate any legal C program into a state machine essentially means giving an operational semantics to the language. Writing an operational semantics is the only practical method of formally specifying the meaning of an arbitrary program written in a language as complicated as C.

The output produced by executing a concurrent program can depend upon how many separate steps are taken when executing a statement like

$$\text{IF } n = 1 \text{ THEN } 1 \text{ ELSE } n * (n-1)!$$

$$\prod_{i=1}^{n} i$$

$$\prod_{i \in \{j \in \mathbf{Z} : 1 \leq j \leq n\}} i$$

Figure 2: Three definitions of $n!$.

$f = i * f$. Errors in concurrent programs often arise because what the programmer thinks of as a single step is actually executed as multiple steps. The programmer may think evaluating $i * f$ is a single step, but in an actual execution a step of a different thread might occur between the reads of i and f.

Most modern multiprocessor computers have weak memory models in which a read or write of i or f is not a single step. These memory models are usually specified in terms of axioms on computations instead of in terms of state machines [3]. We can understand a state machine by mentally executing it, but it is extremely difficult to understand the consequences of a set of axioms. As a result, it is very hard to write multiprocess programs that use unsynchronized reads and writes of shared variables that are correct under such memory models. Instead, we usually program in a way that permits accurate state-machine descriptions of our programs. For example, accesses to shared variables are usually placed in a critical section whose execution can be considered to be a single step. When programmers must use unsynchronized reads and writes, as in operating system code, they seem to use intuitive state machine models of the memory based on a particular computer. Their code often does not work on a later-model computer with a more highly optimized implementation of the same memory model.

Just because Programs 1 and 3 of Figure 1 generate the same sequences of multiplications does not mean that their differences are unimportant. The differences in the other steps they generate may affect their execution speeds. For example, function calls are often more expensive than branches. Moreover, the way a program is written can affect how easy it is to understand, and consequently how easy it is to check its correctness. Consider the three possible definitions of $n!$ in Figure 2. From the first definition, it is more obvious that Program 3 computes 7!. On the other hand, the second

definition makes it is easier to see that Program 1 does. With the third definition (where no ordering of the multiplications is implied), Programs 1 and 2 are most easily seen to compute the correct result and could be considered most similar.

4 Describing State Machines

To use state machines, we need a language for describing them. Any language for describing objects that compute can be viewed as describing state machines, so there is a large choice of possible languages. Every computer scientist will have her favorite—perhaps actor-based or a form of process algebra. I will adopt the one language used in all other branches of science and engineering—namely, mathematics. The formalism underlying this mathematics was developed over a century ago and is considered standard by most mathematicians. It consists of first-order logic and (untyped) set theory.[3] Several formal languages exist for expressing this standard mathematics [2, 18], but I will just use mathematics informally here.

Since a state machine is described by the sets $\mathcal{I}$ and $\mathcal{N}$, we could simply use ordinary set notation to specify these sets. However, there is a simple method of representing states that is used by most scientists and engineers. A set of states is specified by a collection of *state variables* and their *ranges*, which are sets of values. A state s assigns to every variable v a value $s(v)$ in its range. For example, physicists might describe the state of a particle moving in one dimension by variables x (the particle's position) and p (its momentum) whose ranges are the set of real numbers. The state s_t at a time t is described by the real numbers $s_t(x)$ and $s_t(p)$, which physicists usually write $x(t)$ and $p(t)$.

Scientists and engineers specify a subset of the set of states with a *state predicate*, which is a Boolean-valued formula P whose free variables are state variables. We say that a state s *satisfies* a state predicate P iff P equals TRUE when $s(v)$ is substituted for v, for each state variable v. For the particle example with real-valued variables x and p, the state predicate $x = 0$ specifies the set of all states s such that $s(x) = 0$ and $s(p)$ is any real number.

Because most fields of science and engineering study continuous processes, there is no standard way to describe a relation on the set of states. For this purposes, I use a *transition predicate*, which is a Boolean-valued formula

[3] There are several slightly different formulations of standard mathematics, but they are essentially equivalent for scientific and engineering applications.

N whose free variables are primed and/or unprimed state variables. A pair $\langle s, t \rangle$ of states satisfies N iff N equals TRUE when $s(v)$ is substituted for v and $t(v)$ is substituted for v', for each state variable v. For the particle example, $(x' = x + 1) \wedge (p' > x')$ specifies the relation consisting of all pairs $\langle s, t \rangle$ of states such that $t(x) = s(x) + 1$ and $t(p) > t(x)$.

There is a natural isomorphism between state predicates and subsets of the set $\mathcal{S}$ of states, where $P \leftrightarrow \mathcal{P}$ iff $\mathcal{P}$ is the set of states satisfying state predicate P. If $P \leftrightarrow \mathcal{P}$ and $Q \leftrightarrow \mathcal{Q}$, then $P \wedge Q \leftrightarrow \mathcal{P} \cap \mathcal{Q}$ and $P \vee Q \leftrightarrow \mathcal{P} \cup \mathcal{Q}$. Similarly, there is an isomorphism between transition predicates and relations on $\mathcal{S}$, where $\wedge$ corresponds to $\cap$ and $\vee$ to $\cup$ under the isomorphism.

To specify a state machine, we give the state variables and their ranges, and we write a state predicate *Init* and a transition predicate *Next*. I illustrate this with a state machine that describes the following C code, where execution from one label to the next is considered to be a single step.

Program 4

int $i, f = 1;$
test: **if** $(i > 1)$ *mult*: **{** f = $i * f$; $--i$; **goto** *test* ; **}** ;
done:

(The **int** declaration is considered to be a specification of the starting state and not a statement to be executed.) We can describe Program 4 with the variables i, f, and pc, where i and f have as range the set of integers and the range of pc is the set {"test", "mult", "done"} of strings. The predicates *Init* and *Next* are

$$\begin{aligned}
\mathit{Init} \;&\triangleq\; (f = 1) \;\wedge\; (pc = \text{“test”}) \\
\mathit{Next} \;&\triangleq\; \big(\;\; (pc = \text{“test”}) \\
&\qquad \wedge\; (pc' = \text{IF } \; i > 1 \; \text{ THEN } \; \text{“mult”} \; \text{ ELSE } \; \text{“done”}) \\
&\qquad \wedge\; (f' = f) \;\wedge\; (i' = i) \big) \\
&\quad \vee \big(\;\; (pc = \text{“mult”}) \\
&\qquad \wedge\; (pc' = \text{“test”}) \\
&\qquad \wedge\; (f' = i * f) \\
&\qquad \wedge\; (i' = i - 1) \big)
\end{aligned}$$

For this example, let us write a state s as $[i \mapsto s(i), f \mapsto s(f), pc \mapsto s(pc)]$. The set of initial states consists of all states $[i \mapsto i_0,\ f \mapsto 1,\ pc \mapsto \text{“test”}]$ such that i_0 is an integer. The next-state relation includes the pair of states

$$\langle\, [i \mapsto -6,\ f \mapsto 11,\ pc \mapsto \text{“mult”}],\ [i \mapsto -7,\ f \mapsto -66,\ pc \mapsto \text{“test”}] \,\rangle$$

because *Next* equals TRUE under the substitution

$$i \leftarrow -6,\ f \leftarrow 11,\ pc \leftarrow \text{“}\mathsf{mult}\text{”},\ i' \leftarrow -7,\ f' \leftarrow -66,\ pc' \leftarrow \text{“}\mathsf{test}\text{”}$$

The transition predicate *Next* is the disjunction of two formulas, each describing a piece of code that generates a step. The first disjunct describes the **if** test; the second describes the bracketed statement labeled *mult*. Each of these disjuncts is the conjunction of four formulas. Three of them specify the new values of the three variables (their values in the second state) as a function of their old values. The other conjunct contains no primes and is an enabling condition—it is a state predicate that is satisfied by a state s iff there exists a state t such that $\langle s, t \rangle$ satisfies the transition predicate. Such a disjunction of conjunctions is the canonical form of the predicate *Next* in a state-machine specification.

The definition of the *Next* predicate of a state machine is usually built up hierarchically from simpler definitions. For example, we could define the formula *Next* of Program 4 as $Test \lor Mult$, where *Test* and *Mult* are the transition predicates that describe the pieces of code labeled *test* and *mult*, respectively. There can be many ways to decompose the definition of a next-state transition predicate. For a state machine describing a multiprocess algorithm, *Next* is usually defined to have the form $\exists\, p \in P : Proc(p)$, where P is the set of processes and $Proc(p)$ describes the steps of process p. If a programming-language description of the algorithm has a variable x that is local to each process, then the state-machine description has a corresponding variable x whose value is a function with domain P, where $x(p)$ represents the value of process p's copy of the variable.[4]

State-action machines are specified with an initial state predicate *Init* and a state-transition predicate $Next_\alpha$ for each action α. A triple $\langle s, \alpha, t \rangle$ belongs to the next-state set iff $\langle s, t \rangle$ satisfies $Next_\alpha$. The set of all actions is usually partitioned into parameterized sets of actions; for each such set $\{\alpha(i)\}$ one specifies a parameterized state-transition predicate $Next_\alpha(i)$.

There are two standard methods of expressing fairness conditions: as fairness requirements on transition predicates [12] (usually disjuncts of the next-state transition predicate) and as temporal-logic formulas [23]. The temporal logic TLA [18] combines these two approaches by allowing fairness requirements on transition predicates to be written as temporal-logic

[4] In some esoteric formalisms devised by computer scientists, functions are higher-order objects. In standard math, they are just sets of pairs; there is no fundamental difference between a number and a function.

formulas. In fact, it allows the entire specification of a state machine to be written as a single formula.

5 Correctness

I will illustrate how state machines can help reveal fundamental principles by considering what is usually (rather misleadingly) called *correctness*, as in "program correctness" or "proving correctness". The area of correctness encompasses algorithms and systems, not just programs, and it covers writing specifications whose correctness may never be proved.

I will consider only state machines viewed as generators of state behaviors. Most of the concepts introduced can be reformulated for state-action machines as well, but having actions in addition to states makes them more complicated. For simplicity, I will mostly ignore fairness.

Many of the concepts presented here have been explored in the study of action systems and the refinement calculus, but with state machines described using programming language notation [5, 6].

5.1 Invariance

5.1.1 The Inductive Invariant Method

A state predicate is an *invariant* of a state machine iff it is satisfied by all reachable states—that is, by all states that occur in a behavior generated by the state machine. Invariants play an important role in understanding algorithms and programs. For example, a *loop invariant* is a state predicate L that is always true at the beginning of some loop in a program. Predicate L is a loop invariant iff $AtLoop \Rightarrow L$ is an invariant of the program, where *AtLoop* is a state predicate asserting that control is at the beginning of the loop.

A transition predicate T is said to *leave invariant* a state predicate *Inv* iff $Inv \wedge T \Rightarrow Inv'$ holds, where Inv' is the formula obtained from *Inv* by priming every state variable. This condition means that if a state s satisfies *Inv* and $\langle s, t \rangle$ satisfies T, then t satisfies *Inv*. A simple induction argument shows that if *Inv* is implied by the initial predicate *Init* and is left invariant by the next-state predicate *Next*, then *Inv* is an invariant of the state machine. Such an invariant is called an *inductive invariant* of the state machine. The *inductive invariant method* proves that P is an invariant of a state machine by finding an inductive invariant *Inv* that implies P. In

other words, it consists of finding a formula Inv that satisfies

I1. $Init \Rightarrow Inv$
I2. $Inv \wedge Next \Rightarrow Inv'$
I3. $Inv \Rightarrow P$

It is easy to check that if P is an invariant of a state machine, then we obtain an equivalent state machine—that is, one that generates the same behaviors—by replacing the next-state transition predicate $Next$ with $P \wedge Next$ or with $P \wedge Next \wedge P'$.

5.1.2 Composition of Relations and Weakest Preconditions

If $\mathcal{A}$ and $\mathcal{B}$ are relations on $\mathcal{S}$, their composition $\mathcal{A} \cdot \mathcal{B}$ is defined to be the relation consisting of all pairs $\langle s, t \rangle$ of states such that there exists a state u with $\langle s, u \rangle \in \mathcal{A}$ and $\langle u, t \rangle \in \mathcal{B}$. We define composition of transition predicates so it corresponds to composition of relations under the isomorphism between predicates and relations. In other words, we define the composition $A \cdot B$ of transition predicates A and B so that if $A \leftrightarrow \mathcal{A}$ and $B \leftrightarrow \mathcal{B}$, then $A \cdot B \leftrightarrow \mathcal{A} \cdot \mathcal{B}$. Letting x be the tuple $\langle x_1, \ldots, x_n \rangle$ of all state variables and letting v be a tuple $\langle v_1, \ldots, v_n \rangle$ of new variables, we can write

$$A \cdot B \;=\; \exists\, v_1 \in X_1, \ldots, v_n \in X_n \,:\, A[v/x'] \wedge B[v/x]$$

where X_i is the range of x_i, and $A[v/x']$ and $B[v/x]$ are the formulas obtained by substituting v_i for x'_i in A and v_i for x_i in B, for all i.

We define the Kleene star operator for transition predicates by

$$A^* \;\triangleq\; Id \vee A \vee (A \cdot A) \vee (A \cdot A \cdot A) \vee \ldots$$

where Id is the identity predicate $(x'_1 = x_1) \wedge \ldots \wedge (x'_n = x_n)$. Thus, a state pair $\langle s, t \rangle$ satisfies A^* iff there are state pairs $\langle s_1, s_2 \rangle, \ldots, \langle s_{j-1}, s_j \rangle$ satisfying A with $s = s_1$ and $t = s_j$. The transition operator $Init \wedge Next^*$ is satisfied by all state pairs $\langle s, t \rangle$ such that there is a behavior of the state machine that starts in state s and contains state t.

A state predicate is a transition predicate that has no occurrences of primed variables. If A is a transition predicate and P a state predicate, then $A \cdot P$ is the state predicate that is satisfied by a state s iff there is a state t satisfying P such that $\langle s, t \rangle$ satisfies A.

For a state predicate P and action predicate A, we define $wlp(A, P)$ to equal the state predicate $\neg(A \cdot (\neg P))$. (The wlp stands for *weakest liberal precondition* [9].) A state s satisfies $wlp(A, P)$ iff, for every t such that $\langle s, t \rangle$

satisfies A, the state t satisfies P. The following properties of wlp are easy to check, where A and B are transition predicates, P and Q are predicates, and Id is the identity transition predicate.

W1. $wlp(Id, P) \equiv P$

W2. $wlp(A \cdot B, P) \equiv wlp(A, wlp(B, P))$

W3. If $A \Rightarrow B$ then $wlp(B, P) \Rightarrow wlp(A, P)$.

W4. $Q \wedge A \Rightarrow P'$ iff $Q \Rightarrow wlp(A, P)$.

Since Id implies A^*, W1 and W3 imply $wlp(A^*, P) \Rightarrow P$. Since $A \cdot A^*$ implies A^*, W2 and W3 imply $wlp(A^*, P) \Rightarrow wlp(A, wlp(A^*, P))$, which by W4 implies that $wlp(A^*, P)$ is an invariant of A^*. Hence, invariance conditions I2 and I3 (Section 5.1.1) are satisfied when Inv equals $wlp(Next^*, P)$. To prove that P is an invariant of a state machine, we therefore need only prove $Init \Rightarrow wlp(Next^*, P)$.

The ability to prove invariance by proving $Init \Rightarrow wlp(Next^*, P)$ implies that the inductive invariant method is relatively complete. More precisely, if the language for writing state and transition predicates is closed under the operations of composition and taking the Kleene star, and if all valid formulas in the language are provable, then the invariance of any invariant state predicate can be proved by verifying I1–I3 for a suitably chosen inductive invariant Inv.

5.1.3 The Floyd-Hoare Method

The most widely-studied invariance property is partial correctness. A partial correctness property of a program is specified by two state predicates: a *precondition* Pre and a *postcondition* $Post$. The property asserts that if the program is started in a state satisfying Pre and terminates, then its final state satisfies $Post$. Let $Terminated$ be the state predicate asserting that a program is in a final state. The partial correctness property specified by Pre and $Post$ asserts that $Terminated \Rightarrow Post$ is satisfied by all states in all state-machine behaviors that start in a state satisfying Pre. In other words, it is an invariant of the state machine obtained by changing the initial predicate to $Init \wedge Pre$.

The Floyd-Hoare method for proving partial correctness [11, 16] annotates the program text by attaching a state predicate P_c to each control point c. It is a special case of the inductive invariant method in which the inductive invariant Inv is the predicate $\forall\, c \in C : (pc = c) \Rightarrow P_c$, where C is

the set of all control points and $pc = c$ asserts that control is at c. For pre- and postconditions *Pre* and *Post*, condition I1 reduces to $Init \wedge Pre \Rightarrow P_{ini}$ and I3 reduces to $P_{fin} \Rightarrow Post$, where *ini* is the initial control point and *fin* is the final control point. To prove I2, the transition predicate *Next* is written $\exists\, c \in Next_c$, where $Next_c$ describes the operation at control point c. Condition I2 then reduces to verifying $Inv \wedge Next_c \Rightarrow Inv'$ for each control point c.

For example, suppose the program has variables x, y, and z and contains the statement

$$c\text{: } x = y + 1\,;\ d\text{: } \ldots$$

Then $Next_c$ equals

$$(pc = c)\ \wedge\ (pc' = d)\ \wedge\ (x' = y + 1)\ \wedge\ (y' = y)\ \wedge\ (z' = z)$$

and the condition $Inv \wedge Next_c \Rightarrow Inv'$ reduces to

$$P_c\ \wedge\ (x' = y + 1)\ \wedge\ (y' = y)\ \wedge\ (z' = z)\ \Rightarrow\ P'_d \tag{1}$$

If P_c and P_d do not mention pc, then this condition is equivalent to

$$P_c[y + 1/x] \Rightarrow P_d$$

where $P_c[y{+}1/x]$ is P_c with $y{+}1$ substituted for x. This latter formula is the standard Floyd-Hoare verification condition for the assignment statement $x = y + 1$.

If we replace $x = y + 1$ by an arbitrary statement $\mathbf{S}$, then (1) becomes $P_c \wedge S \Rightarrow P'_d$, where S is the transition predicate that represents $\mathbf{S}$. This formula is the meaning of the Hoare triple $\{P_c\}\mathbf{S}\{P_d\}$. In general, decomposing condition I2 in this way for an arbitrary program yields the verification conditions for the various kinds of statements in the programming language.

The relative completeness of the inductive invariant method, discussed in Section 5.1.2 above, implies standard results about the relative completeness of the Floyd-Hoare method for simple programming languages [4]. The general completeness result for state machines assumes that one can write predicates that describe the entire state. However, because computer scientists are so fixated on languages, they often consider approaches like the simple Floyd-Hoare method whose only state predicates are ones that can be expressed in the programming language. Thus, they write state predicates using only program variables and cannot mention parts of the state like *pc*. For very simple sequential programming languages, the Floyd-Hoare method is relatively complete even though its state predicates mention only

program variables. However, it is incomplete for a programming language with procedures, since completeness requires the use of state predicates that describe the calling stack.

For concurrent programs, we are interested in a richer class of invariance properties. For example, mutual exclusion is the invariance of the state predicate asserting that two different processes are not both in their critical sections. The Owicki-Gries method [22] generalizes the Floyd-Hoare method to concurrent programs. It involves assigning a state predicate P_c^p to each control point c of every process p. It is an instance of the inductive invariant method in which the inductive invariant Inv is the conjunction of all formulas $(pc(p) = c) \Rightarrow P_c^p$, where $pc(p)$ is the current control point of process p's program. Using a similar decomposition of the next-state transition predicate $Next$, condition I2 reduces to the Owicki-Gries method's "sequential consistency" and "interference freedom" conditions. However, even for the simplest concurrent programming languages, this method is incomplete if the state predicates P_c^p cannot mention pc. With pc considered politically incorrect, Owicki and Gries added dummy variables to the program to allow the control state to be mentioned in state predicates. This makes the Owicki-Gries method complete for simple programming languages, since pc can be introduced as a dummy variable.

The Owicki-Gries method is very easy to derive and to understand as an instance of the inductive invariant method, as long as one can talk about program control. However, it becomes rather mysterious when one refuses to do so because the programming language doesn't have a way of expressing it. Dijkstra demonstrated how complicated the method can then appear [10].

5.2 Refinement

The concept of a computing object $\mathcal{Y}$ refining another computing object $\mathcal{X}$ occurs in various guises. We sometimes say that $\mathcal{Y}$ implements $\mathcal{X}$, and we may call $\mathcal{X}$ and $\mathcal{Y}$ the specification and the implementation, the abstract system and the concrete system, or the higher-level and lower-level specifications. The most common example is $\mathcal{X}$ a program in a higher-level language and $\mathcal{Y}$ its compiled version. As in the case of compilation, we may start with $\mathcal{X}$ and refine it to obtain $\mathcal{Y}$. We may also start with $\mathcal{Y}$ and derive $\mathcal{X}$ as a higher-level or more abstract view of $\mathcal{Y}$. For example, we may explain an algorithm $\mathcal{Y}$ by finding a more abstract algorithm $\mathcal{X}$ from which $\mathcal{Y}$ can be derived.

For $\mathcal{Y}$ to refine $\mathcal{X}$, there must be a notion of a computation c_y of $\mathcal{Y}$ refining a computation c_x of $\mathcal{X}$. Let's write $c_y \propto c_x$ to mean that c_y refines

c_x. There are two basic notions of refinement. The first requires that $\mathcal{X}$ and $\mathcal{Y}$ be equivalent under refinement, so $\propto$ is a 1-1 correspondence between the computations of $\mathcal{Y}$ and of $\mathcal{X}$. A popular example of this notion of refinement is *bisimulation* [21]. The second concept of refinement is that for every computation c_y of $\mathcal{Y}$ there is a computation c_x of $\mathcal{X}$ with $c_y \propto c_x$.

Both concepts of refinement have their uses. However, for discussing correctness, the second is the more appropriate one. Since we are here taking computations to be state behaviors, we define a relation $\propto$ on behaviors to be a *refinement* of $\mathcal{X}$ by $\mathcal{Y}$ iff for every behavior σ generated by $\mathcal{Y}$ there is a behavior τ generated by $\mathcal{X}$ such that $\sigma \propto \tau$. We now consider how the relation $\propto$ is defined.

5.2.1 Data Refinement

Data refinement means replacing the data types used to describe the state machine $\mathcal{X}$ by different data types—usually lower-level or more concrete ones. For example, $\mathcal{X}$ might be defined in terms of a queue q that in $\mathcal{Y}$ is represented by an array Arr, a pointer ptr to the queue's first element, and the queue's length len.

A data refinement is specified by an *abstraction relation* $\mathcal{R}$ from the state set $\mathcal{S}_\mathcal{Y}$ of $\mathcal{Y}$ to the state set $\mathcal{S}_\mathcal{X}$ of $\mathcal{X}$. In other words, $\mathcal{R} \subseteq \mathcal{S}_\mathcal{Y} \times \mathcal{S}_\mathcal{X}$. If σ is the behavior $s_1 \rightarrow s_2 \rightarrow \ldots$ in $\mathcal{S}_\mathcal{Y}$ and τ is the behavior $t_1 \rightarrow t_2 \rightarrow \ldots$, then we define the relation $\propto_\mathcal{R}$ on behaviors by $\sigma \propto_\mathcal{R} \tau$ iff σ and τ have the same length and $\langle s_i, t_i \rangle \in \mathcal{R}$ for all i. We say that $\mathcal{R}$ is a *data refinement* of $\mathcal{X}$ by $\mathcal{Y}$ iff $\propto_\mathcal{R}$ is a refinement of $\mathcal{X}$ by $\mathcal{Y}$.

Let $x_1, \ldots, x_n$ be the state variables of $\mathcal{X}$ and $y_1, \ldots, y_m$ be the state variables of $\mathcal{Y}$, and assume that the x_i and y_j are all distinct. The relation $\mathcal{R}$ is then specified by a predicate R whose free variables are the x_i and y_j. If $\mathcal{X}$ has no fairness conditions, then $\mathcal{R}$ is a data refinement if the following two conditions are satisfied, where subscripts are used in the obvious way to name the state machines' defining predicates.

R1. $Init_\mathcal{Y} \;\wedge\; R \;\Rightarrow\; Init_\mathcal{X}$

R2. $Next_\mathcal{Y} \;\wedge\; R \;\wedge\; R' \;\Rightarrow\; Next_\mathcal{X}$

As observed in Section 5.1.1, if Inv is an invariant of state machine $\mathcal{Y}$, then we can replace $Next_\mathcal{Y}$ by $Inv \wedge Next_\mathcal{Y} \wedge Inv'$.

An important special case is when the abstraction relation $\mathcal{R}$ is a function. In that case, $\mathcal{R}$ is specified by n functions $\overline{x}_i$ of m arguments, where R equals

$$(x_1 = \overline{x}_1(y_1, \ldots, y_m)) \;\wedge\; \ldots \;\wedge\; (x_n = \overline{x}_n(y_1, \ldots, y_m))$$

Let $\overline{F}$ be the predicate obtained from a predicate F by substituting $x_i \leftarrow \overline{x}_i(y_1, \ldots, y_m)$ and ${x'}_i \leftarrow \overline{x}_i(y'_1, \ldots, y'_m)$, for each i. Conditions R1 and R2 then become

$$\begin{array}{ll} \text{R1}_f. & \mathit{Init}_{\mathcal{Y}} \;\Rightarrow\; \overline{\mathit{Init}_{\mathcal{X}}} \\ \text{R2}_f. & \mathit{Next}_{\mathcal{Y}} \;\Rightarrow\; \overline{\mathit{Next}_{\mathcal{X}}} \end{array}$$

When written this way, the abstraction function is called a *refinement mapping* [1]. In the example of refining a queue q by variables Arr, ptr, and len, the refinement mapping is defined so that $\overline{q}(A, p, l)$ equals the contents of the queue corresponding to $Arr = A$, $ptr = p$, and $len = l$.

When the state machines have fairness conditions, to prove refinement we must also show that the fairness conditions of $\mathcal{Y}$ imply the fairness conditions of $\mathcal{X}$. In general, this is easier to do for a refinement mapping than for an arbitrary data refinement.

5.2.2 An Example of Data Refinement

Starting from a state machine $\mathcal{X}$, we can derive a state machine $\mathcal{Y}$ that refines it by defining a suitable refinement mapping. As an instructive example, we derive a simple but important algorithm for alternately executing two operations $\mathcal{A}$ and $\mathcal{B}$ from a trivial algorithm. The trivial algorithm might be written in a programming language as:

loop $a{:}\,\mathcal{A}\,;\; b{:}\,\mathcal{B}$ **endloop**

For simplicity, let's suppose that operations $\mathcal{A}$ and $\mathcal{B}$ access only a single variable x. This algorithm can be written as a state machine $\mathcal{X}$ with

$$\begin{array}{lcl} \mathit{Init}_{\mathcal{X}} & \triangleq & (pc = a) \,\wedge\, (x = x_0) \\ \mathit{Next}_{\mathcal{X}} & \triangleq & \quad ((pc = a) \,\wedge\, A \,\wedge\, (pc' = b)) \\ & & \vee\; ((pc = b) \,\wedge\, B \,\wedge\, (pc' = a)) \end{array}$$

for suitable transition predicates A and B that mention only x and for some initial value x_0.

Let us now refine the variable pc with two variables p and c whose range is the set $\{0, 1\}$, defining the refinement mapping by

$$\begin{array}{lcl} \overline{pc} & \triangleq & \text{IF}\;\; p = c \;\; \text{THEN}\;\; a \;\; \text{ELSE}\;\; b \\ \overline{x} & \triangleq & x \end{array} \tag{2}$$

Since $\overline{pc} = a$ iff $p = c$, and $\overline{pc} = b$ iff $p \neq c$, and since A and B mention only x, we have

$$\begin{aligned}
\overline{Init_{\mathcal{X}}} &\triangleq (p = c) \wedge (x = x_0) \\
\overline{Next_{\mathcal{X}}} &\triangleq \quad ((p = c) \wedge A \wedge (p' \neq c')) \\
&\quad \vee \ ((p \neq c) \wedge B \wedge (p' = c'))
\end{aligned}$$

These predicates define a state machine with variables p, c, and x. However, $\overline{Next_{\mathcal{X}}}$ does not have the canonical form of a next-state transition predicate because its two disjuncts do not specify the values of p' and c' as functions of p and c. We obtain our algorithm $\mathcal{Y}$ be defining a canonical-form transition predicate $Next_{\mathcal{Y}}$ that implies $\overline{Next_{\mathcal{X}}}$.

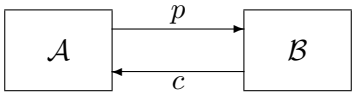
Let $\oplus$ be addition modulo 2, so $1 \oplus 1 = 0$. Since p and c can equal only 0 or 1, we have $p \neq c$ iff $p = c \oplus 1$ or, equivalently, $c = p \oplus 1$. Each disjunct of $\overline{Next_{\mathcal{X}}}$ is therefore satisfied iff either $p' = p \oplus 1$ and $c' = c$, or $p' = p$ and $c' = c \oplus 1$. The following transition predicate $Next_{\mathcal{Y}}$ thus implies $\overline{Next_{\mathcal{X}}}$:

$$\begin{aligned}
Next_{\mathcal{Y}} \triangleq \quad & ((p = c) \wedge A \wedge (p' = p \oplus 1) \wedge (c' = c)) \\
\vee \ & ((p \neq c) \wedge B \wedge (p' = p) \wedge (c' = c \oplus 1))
\end{aligned}$$

Let $\mathcal{Y}$ be the state machine with this next-state transition predicate and with initial predicate $Init_{\mathcal{Y}}$ equal to $\overline{Init_{\mathcal{X}}}$.

We defined $Next_{\mathcal{Y}}$ so it would satisfy $\mathrm{R2}_f$, and $\mathrm{R1}_f$ trivially holds (because $Init_{\mathcal{Y}}$ is defined to equal $\overline{Init_{\mathcal{X}}}$). Hence $\mathcal{Y}$ refines $\mathcal{X}$ under the refinement mapping (2). Since the steps of a behavior of $\mathcal{X}$ alternately satisfy A and B, and the refinement mapping leaves x unchanged, we deduce that steps of $\mathcal{Y}$ also alternately satisfy A and B. Thus, like $\mathcal{X}$, the state machine $\mathcal{Y}$ describes an algorithm for alternately executing $\mathcal{A}$ and $\mathcal{B}$ operations.

Algorithm $\mathcal{Y}$ is an important hardware protocol called *two-phase handshake* [20]. A device that performs operation $\mathcal{A}$ is joined to one that performs $\mathcal{B}$ by two wires whose levels are represented by the values of p and c.



The first device performs an $\mathcal{A}$ operation when the levels of the two wires are the same and complements the level of p. The second performs a $\mathcal{B}$ operation when the levels of the two wires are different and complements the level of c.

The key step in this derivation was the substitution of the expression $\overline{pc}$ for the variable pc in the initial and next-state predicates. Such substitution is impossible in most languages for describing state machines, including programming languages. There is no way to substitute for pc in the

programming-language representation of $\mathcal{X}$, since *pc* doesn't appear explicitly. Even if *pc* were described by a variable, it would make no sense to substitute the expression $\overline{pc}$ for *pc* in a statement such as $pc := b$.

5.2.3 Refinement with Stuttering

In data refinement, behaviors of state machine $\mathcal{X}$ and the corresponding behaviors of its refinement $\mathcal{Y}$ have the same number of steps. More often, a single step of $\mathcal{X}$ is refined to a sequence of steps of $\mathcal{Y}$. As an example, suppose $\mathcal{X}$ is a state machine representing a simple hour clock, described by a variable *hour* that cycles through the values 1, 2, ..., 12.[5] A simple refinement is a state machine $\mathcal{Y}$ representing an hour-minute clock, with variables *hr* and *min*, in which *min* cycles through the values 0, 1, ..., 59, and *hr* is incremented when *min* changes from 59 to 0.

If we ignore the minute, then an hour-minute clock is just an hour clock. So, we expect $\mathcal{Y}$ to implement $\mathcal{X}$ under the simple data abstraction defined by the refinement mapping $\overline{hour} = hr$. However, if σ is a behavior of the hour-minute clock $\mathcal{Y}$, then $\sigma \propto \tau$ only for a behavior τ with 59 steps that leave the state unchanged between every step that increments *hour*—steps that are called *stuttering* steps. The state machine $\mathcal{X}$ does not generate such stuttering steps.

To describe this kind of refinement, we define the relation $\sim$ on state behaviors by $\sigma \sim \tau$ iff τ can be obtained from σ by adding or deleting stuttering steps. We can then define the refinement $\overset{\propto}{\sim}_{\mathcal{R}}$ for an abstraction relation $\mathcal{R}$ by $\sigma \overset{\propto}{\sim}_{\mathcal{R}} \tau$ iff there exists a state behavior ρ such that $\sigma \propto_{\mathcal{R}} \rho$ and $\rho \sim \tau$. If $\mathcal{R}$ is defined by the refinement mapping $\overline{hour} = hr$, then $\overset{\propto}{\sim}_{\mathcal{R}}$ is a refinement of the hour state machine by the hour-minute state machine. In such a case, we say that $\mathcal{R}$ is a data refinement *with stuttering*.

In Section 2.4, we saw that we sometimes want to specify a system with a state machine in which some of the state is considered to be hidden. This is usually done by letting the machine's state set $\mathcal{S}$ be the Cartesian product $\mathcal{S}^v \times \mathcal{S}^h$ of a set $\mathcal{S}^v$ of visible (also called external) states and a set $\mathcal{S}^h$ of hidden (also called internal) states. If $\mathcal{X}$ and $\mathcal{Y}$ are two such specifications with the same set $\mathcal{S}^v$ of visible states, then $\mathcal{Y}$ is said to *implement* $\mathcal{X}$ iff every behavior of $\mathcal{Y}$ has the same visible states as some behavior of $\mathcal{X}$. If $\mathcal{V}$ is the abstraction relation defined by $\langle\langle v, h\rangle, \langle w, j\rangle\rangle \in \mathcal{V}$ iff $v = w$, then $\mathcal{Y}$ implements $\mathcal{X}$ iff $\mathcal{V}$ is a data refinement with stuttering of $\mathcal{X}$ by $\mathcal{Y}$.

Proving data refinement with stuttering is similar to proving ordinary data refinement. In condition R2 or $\mathrm{R2}_f$ of Section 5.2.1, we just replace

[5] We are ignoring the physical time that elapses between ticks.

$Next_{\mathcal{X}}$ by $Next_{\mathcal{X}} \vee Id_{\mathcal{X}}$, where $Id_{\mathcal{X}}$ equals $(x'_1 = x_1) \wedge \cdots \wedge (x'_n = x_n)$ and the x_i are the state variables of $\mathcal{X}$. However, stuttering can complicate the proof of the fairness conditions of $\mathcal{X}$.

In a wide variety of situations, from compilation to implementing a protocol by a distributed algorithm, refinement is data refinement with stuttering. In most of these cases, the data refinement is defined by a refinement mapping.

5.2.4 Invariance Under Stuttering

We can reduce data refinement with stuttering to ordinary data refinement by simply requiring that a state machine's next-state predicate allow stuttering steps. For example, the refinement mapping $\overline{hour} = hr$ is a data refinement of the hour clock by the hour-minute clock because the hour clock's state machine generates behaviors that contain 59 (or more) stuttering steps between changes to *hour*.

Requiring the next-state predicate to allow stuttering steps is a special case of the general principle of invariance under stuttering[6], which is that we should never distinguish between two behaviors that differ only by stuttering steps [17]. The idea behind this principle is that, in a state-based approach, the state completely describes a system. If the state doesn't change, then nothing has happened. Therefore, two behaviors that differ only by stuttering steps represent the same computation.

If the next-state relation $\mathcal{N}$ allows stuttering steps, then $\langle s, s \rangle$ is in $\mathcal{N}$ for all states s. Condition S3 in the definition of the behaviors generated by a state machine (Section 2.2) then implies that every behavior is infinite. A behavior that ends in an infinite sequence of stuttering steps is one in which the computing object represented by the state machine has halted. To disallow premature halting, we must add a fairness condition to disallow behaviors that stutter forever in a state in which a non-stuttering step is possible. (The requisite condition is weak fairness of the transition predicate $Next \wedge \neg Id$, where Id is the predicate corresponding to the identity relation on the set of states.)

Invariance under stuttering simplifies reasoning about refinement. However, it may not be a good idea for other applications.

[6] This use of *invariance* is unrelated to its use in Section 5.1

6 Conclusion

I have tried to show that state machines provide a simple conceptual framework for describing computation. Some of the unification provided by this framework may have passed unnoticed—for example, that there is no fundamental difference between termination and deadlock. Much of the discussion has been quite superficial. For example, nothing was said about computational complexity except that it measures the number of steps in a behavior. To measure complexity accurately enough so constant factors matter, one must decide what operations should be counted. When computers were slower, one counted floating point multiplications and divisions, but not additions and subtractions; today, memory references are usually what matter. Deciding what operations to count means choosing what constitutes an individual step of the state machine.

I have gone into some detail only in the area of correctness. Even there, much more can be said. For example, Hoare logic [16] was only touched upon. It can be explained as a way of considering a state machine $\mathcal{Y}$ to be a refinement of a high-level state machine $\mathcal{X}$ whose behaviors consist of at most one step, where $s \to t$ is a behavior of $\mathcal{X}$ iff s is an initial state and t a final state of $\mathcal{Y}$. The next-state predicate of $\mathcal{X}$ is $Next^*_{\mathcal{Y}} \land Terminated'_{\mathcal{Y}}$, where $Next_{\mathcal{Y}}$ is the next-state predicate of $\mathcal{Y}$ and $Terminated_{\mathcal{Y}}$ is true iff $\mathcal{Y}$ has terminated.

I have used mathematics in a simple, naive way to specify state machines. The one non-obvious idea that I mentioned is stuttering invariance (Section 5.2.4). There are other sophisticated ideas that simplify the mathematics of state machines. One is to use a single state space for all state machines, in which a state is an assignment of values to all possible variables. Variables not mentioned in the initial and next-state predicates can assume any values. This mirrors ordinary mathematics, in which writing $x + y - 1$ does not imply the non-existence of the variable z. The use of a single state space makes it easier to relate different state machines and to combine them in ways such as parallel composition.

Many languages are expressive enough to describe any state machine and could thus also provide a uniform framework for describing computation. The advantage of state machines is that they can be described using ordinary mathematics. Mathematics is simpler and more expressive than any language I know that computer scientists have devised to describe computations. It is the basis of all other branches of science and engineering. Mathematics should provide the foundation for computer science as well.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, 1996.

[3] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, Boston, third edition, 1998.

[4] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—part one. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[5] R. J. R. Back. Refinement calculus, part ii: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, May/June 1989.

[6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[7] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[8] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Enginnering and Methodology*, 2(1):1–46, 1993.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[10] Edsger W. Dijkstra. A personal summary of the Gries-Owicki theory. In Edsger W. Dijkstra, editor, *Selected Writings on Computing: A Personal Perspective*, chapter EWD554, pages 188–199. Springer-Verlag, New York, Heidelberg, Berlin, 1982.

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

[12] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.

[13] Yuri Gurevich. Evolving algebra 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[14] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[15] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[16] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[17] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.

[18] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at `http://lamport.org`.

[19] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Symposium on the Principles of Distributed Computing*, pages 137–151. ACM, August 1987.

[20] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, Massachusetts, 1980.

[21] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[22] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–284, May 1976.

[23] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[24] Vaughan R. Pratt. Transition and cancellation in concurrency and branching time. *Mathematical Structures in Computer Science*, 13(4):485–529, 2003.

[25] Richard L. Schwartz and P. M. Melliar-Smith. Temporal logic specification of distributed systems. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 446–454. IEEE Computer Society Press, April 1981.

Appendix: The Definition of $\pi(\sigma)$

To define the set of pomsets generated by state-action machine, we now define the pomset $\pi(\sigma)$ corresponding to a computation σ of the machine. This definition was mentioned by Pratt [24, Section 2.2].

Let σ be the state-action computation

$$s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \xrightarrow{\alpha_3} \cdots$$

For simplicity, assume that σ is an infinite sequence. Modifying the definitions to apply to finite computations is straightforward but somewhat tedious.

We begin by defining $\sigma \overset{i}{\Rightarrow} \tau$ for a positive integer i and a computation τ to mean that τ can be obtained from σ by interchanging α_i with α_{i-1}. More precisely, $\sigma \overset{i}{\Rightarrow} \tau$ is true (for $i > 1$) iff $\alpha_i \neq \alpha_{i-1}$ and τ is the same as σ except with $s_{i-1} \xrightarrow{\alpha_{i-1}} s_i \xrightarrow{\alpha_i} s_{i+1}$ replaced by $s_{i-1} \xrightarrow{\alpha_i} t \xrightarrow{\alpha_{i-1}} s_{i+1}$ for some state t.

We next define the relation $i \leftrightarrow_\sigma j$ to mean that we can obtain computations of the state-action machine by interchanging α_j with α_{j-1}, then with α_{j-2}, ..., then with α_i. More precisely, we inductively define $i \leftrightarrow_\sigma j$ to hold iff either

- $i = j$ or
- $i < j$ and $\sigma \overset{j}{\Rightarrow} \tau$ holds for some computation τ of the state machine with $i \leftrightarrow_\tau (j-1)$.

Finally, we define $\pi(\sigma)$ to be the set $\{e_1, e_2, \ldots\}$, where each e_i is labeled by α_i, with the partial order $\prec$ defined by $e_i \prec e_j$ iff $i < j$ and $i \leftrightarrow_\sigma j$ does *not* hold.