Interactive Algorithms 2005

Yuri Gurevich
Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA

Technical Report MSR-TR-2005-73, June 5, 2005

Abstract

A sequential algorithm just follows its instructions and thus cannot make a nondeterministic choice all by itself, but it can be instructed to solicit outside help to make a choice. Similarly, an object-oriented program cannot create a new object all by itself; a create-a-new-object command solicits outside help. These are but two examples of intra-step interaction of an algorithm with its environment. Here we motivate and survey recent work on interactive algorithms within the Behavioral Computation Theory project.

Contents

1 Introduction

In 1982, the University of Michigan hired this logician on his promise to become a computer scientist. The logician eagerly wanted to become a computer scientist. But what is computer science? Is it really a science? What is it about?

After thinking a while, we concluded that computer science is largely about algorithms. Operating systems, compilers, programming languages, etc. are all algorithms, in a wide sense of the word. For example, a programming language can be seen as a universal algorithm that applies the given program to the given data. In practice, you may need a compiler and a machine to run the compiled program on, but this is invisible on the abstraction level of the programming language.

A problem arises: What is an algorithm? To us, this is a fundamental problem of computer science, and we have been working on it ever since.

But didn't Turing solve the problem? The answer to this question depends on how you think of algorithms. If all you care is the input-to-output function of the algorithm, then yes, Turing solved the problem. But the behavior of an algorithm may be much richer than its input-to-output function. An algorithm has its natural abstraction level, and the data structures employed by an algorithm are intrinsic to its behavior. The parallelism of a parallel algorithm is an inherent part of its behavior. Similarly, the interactivity of an interactive algorithm is an inherent part of its behavior as well.

Is there a solution à la Turing to the problem what an algorithm is? In other words, is there a state-machine model that captures the notion of algorithm up to behavioral equivalence? Our impression was, and still is, that the answer is yes. In [13], we defined sequential abstract state machines (ASMs) and put forward a sequential ASM thesis: for every sequential algorithm, there is a sequential ASM with the same behavior. In particular, the ASM is supposed to simulate the given algorithm step-for-step. In [14], we defined parallel and distributed abstract state machines and generalized the ASM thesis for parallel and distributed algorithms. Parallel ASMs gave rise to a specification (and high-level programming) language AsmL [2] developed by the group of Foundations of Software Engineering of Microsoft Research.

At this point, the story forks. One branch leads to experimental evidence for the ASM thesis and to applications of ASMs [1, 2, 12]. Another branch leads to behavioral computation theory. We take the second branch here and restrict attention to *sequential time algorithms* that compute in a sequence of discrete steps.

In § 2 we discuss a newer approach to the explication of the notion of algorithm. The new approach is axiomatic, but it also involves a machine characterization of algorithms. This newer approach is used in the rest of the article.

In § 3 we sketch our explication of sequential (or small-step) algorithms [15]. We mention also the explication of parallel (or wide-step) algorithms in [3] but briefly. In either case, the algorithms in questions are *isolated step algorithms* that abstain from intra-step interaction with the environment. They can interact with the environment in the inter-step manner, however.

§ 4 is a quick introduction to the study of intra-step interaction of an algorithm with its environment; much of the section reflects [5]. We motivate the study of intra-step interaction and attempt to demonstrate how ubiquitous intra-step interaction is. Numerous disparate phenomena are best understood as special cases of intra-step interaction. We discuss various forms of intra-step interaction, introduce the query mechanism of [5] and attempt to demonstrate the universality of the query mechanism: the atomic interactions of any mechanism are queries. In the rest of the article, we concentrate on intra-step interaction; by default interaction means intra-step interaction. To simplify the exposition, we consider primarily the small-step (rather than wide-step) algorithms; by default algorithms are small-step algorithms.

§ 5 is devoted to the explication of *ordinary interactive algorithms* [5, 6, 7]. Ordinary algorithms never complete a step until all queries from that step have been answered. Furthermore, the only information from the environment that an ordinary algorithm uses during a step is answers to its queries.

§ 6 is devoted to the explication of general interactive algorithms [8, 9, 10]. Contrary to ordinary interactive algorithms, a general interactive algorithm can be *impatient* and complete a step without waiting for all queries from that step to have been answered. It also can be *time sensitive*, so that its actions during a step depend not only on the answers to its queries but also on the order in which the answers have arrived. We mention also the explication of general wide-step algorithms [11] but briefly.

§ 7 is a concluding remark.

Much of this article reflects joint work with Andreas Blass, Benjamin Rossman and Dean Rosenzweig.

2 Explication of Algorithms

The theses mentioned in the introduction equate an informal, intuitive notion with a formal, mathematical notion. You cannot prove such a thesis mathematically but you can argue for it. Both Church and Turing argued for their theses. While their theses are equivalent, their arguments were quite different [4]. The ASM theses, mentioned in the introduction, have the following form.

ASM Thesis Form

1. Describe informally a class **A** of algorithms.

2. Describe the behavioral equivalence of **A** algorithms. Intuitively two algorithms are behaviorally equivalent if they do the same thing in all circumstances. Since **A** is defined informally, the behavioral equivalence may be informal as well.

3. Define a class **M** of abstract state machines.

4. Claim that $\mathbf{M} \subseteq \mathbf{A}$ and that every $A \in \mathbf{A}$ is behaviorally equivalent to some $M \in \mathbf{M}$.

The thesis for a class **A** of algorithms explicates algorithms in **A** as abstract state machines in **M**. For example, sequential algorithms are explicated as sequential ASMs. The thesis is open to criticism. One can try to construct an ASM in **M** that falls off **A** or an algorithm in **A** that is not behaviorally equivalent to any ASM in **M**.

Since the ASM thesis for **A** cannot be proven mathematically, experimental confirmation of the thesis is indispensable; this partially explains the interest in applications of ASMs in the ASM community. But one can argue for the thesis, and we looked for the best way to do that. Eventually we arrived at a newer and better explication procedure.

Algorithm Explication Procedure

1. Axiomatize the class **A** of the algorithms of interest. This is the hardest part. You try to find the most convincing axioms (or postulates) possible.

2. Define precisely the notion of behavioral equivalence. If there is already an ASM thesis T for **A**, you may want to use the behavioral

equivalence of T or a precise version of the behavioral equivalence of T.

3. Define a class $\mathbf{M}$ of abstract state machines. If there is already an ASM thesis T for $\mathbf{A}$, you may want to use the abstract state machines of T.

4. Prove the following characterization theorem for $\mathbf{A}$: $\mathbf{M} \subseteq \mathbf{A}$ and every $A \in \mathbf{M}$ is behaviorally equivalent to some $M \in \mathbf{M}$.

The characterization provides a theoretical programming language for $\mathbf{A}$ and opens a way for more practical languages for $\mathbf{A}$. Any instance of the explication procedure is open to criticism of course. In particular, one may criticize the axiomatization and the behavioral equivalence relation.

If an explication procedure for $\mathbf{A}$ uses (a precise version of) the behavioral equivalence and the machines of the ASM thesis for $\mathbf{A}$, then the explication procedure can be viewed as a proof of the thesis given the axiomatization.

A priori it is not obvious at all that a convincing axiomatization is possible. But our experience seems to be encouraging. The explication procedure was used for the first time in [15] where sequential algorithms were axiomatized and the sequential ASM thesis proved; see more about that in the next section. In [3], parallel algorithms were axiomatized and the parallel ASM thesis was proved, except that we slightly modified the notion of parallel ASM. Additional uses of the explication procedure will be addressed in § 4–6.

In both, [15] and [3], two algorithms are behaviorally equivalent if they have the same states, initial states and transition function. It follows that behaviorally equivalent algorithms simulate each other step-for-step. We have been criticized that this behavioral equivalence is too fine, that step-for-step simulation is too much to require, that appropriate bisimulation may be a better behavioral equivalence. We agree that in some applications bisimulation is the right equivalence notion. But notice this: the finer the behavioral equivalence, the stronger the characterization theorem.

3 Isolated-Step Algorithms

As we mentioned above, sequential algorithms were explicated in [15]. Here we recall and motivate parts of that explication needed to make our story self-contained.

Imagine that you have some entity E. What does it mean that E is a sequential algorithm? A part of the answer is easy: every algorithm is a (not necessarily finite-state) automaton.

Postulate 3.1 (Sequential Time). *The entity E determines*

- *a nonempty collection of states,*
- *a nonempty collection of initial states, and*
- *a state-transition function.*

The postulate does not say anything about final states; we refer the interested reader to [15, § 3.3.2] in this connection. This single postulate allows us to define behavioral equivalence of sequential algorithms.

Definition 3.2. Two sequential algorithms are *behaviorally equivalent* if they have the same states, initial states and transition function.

It is harder to see what else can be said about sequential algorithms in full generality. Of course, every algorithm has a program of one kind or another, but we don't know how to turn this into a postulate or postulates. There are so many different programming notations in use already, and it is bewildering to imagine all possible programming notations.

Some logicians, notably Andrey A. Markov [17], insisted that the input to an algorithm should be *constructive*, like a string or matrix, so that you can actually write it down. This excludes abstract finite graphs for example. How would you put an abstract graph on the Turing machine tape? It turned out, however, that the constructive input requirement is too restrictive. Relational databases for example represent abstract structures, in particular graphs, and serve as inputs to important algorithms.

Remark 3.3. You can represent an abstract graph by an adjacency matrix. But this representation is not unique. Note also that it is not known whether there is a polynomial-time algorithm that, given two adjacency matrices, determines whether they represent the same graph.

A characteristic property of sequential algorithms is that they change their state only locally in any one step. Andrey N. Kolmogorov, who looked into this problem, spoke about "steps whose complexity is bounded in advance" [16]. We prefer to speak about bounded work instead; the amount of work done by a sequential algorithm in any one step is bounded, and the bound depends only on the algorithm and not on the state or the input. But we don't know how to measure the complexity of a step or the work done during a step. Fortunately we found a way around this difficulty. To this end, we need two additional postulates.

According to the abstract state postulate, all states of the entity E are structures (that is first-order structures) of a fixed vocabulary. If X is an (initial) state of A and a structure Y is isomorphic to X then Y is an (initial) state of A. The abstract state postulate allows us to introduce an abstract notion of location and to mark locations explored by an algorithm during a given step. The bounded exploration postulate bounds the number of locations explored by an algorithm during any step; the bound depends only on the algorithm and not on the state or the input. See details in [15].

Definition 3.4. A sequential algorithm is any entity that satisfies the sequential-time, abstract-state and bounded-exploration postulates.

A *sequential abstract state machine* is given is by a program, a nonempty isomorphism-closed collection of states and a nonempty isomorphism-closed subcollection of initial states. The program determines the state transition function.

Like a Turing machine program, a sequential ASM program describes only one step of the ASM. It is presumed that this step is executed over and over again. The machine halts when the execution of a step does not change the state of the machine. The simplest sequential ASM programs are assignments:

$$f(t_1, \ldots, t_j) := t_0$$

Here f is a j-ary *dynamic function* and every t_i is a ground first-order term. To execute such a program, evaluate every t_i at the given state; let the result be a_i. Then set the value of $f(a_1, \ldots, a_j)$ to a_0. Any other sequential ASM program is constructed from assignments by means of two constructs: if-then-else and do-in-parallel. Here is a sequential ASM program for the Euclidean algorithm: given two natural numbers a and b, it computes their greatest common divisor d.

Example 3.5 (Euclidean Algorithm 1).

```
if a = 0 then d := b
else do in-parallel
   a := b mod a
   b := a
```

The do-in-parallel constructs allows us to compose and execute in parallel two or more programs. In the case when every component is an assignment, the parallel composition can be written as a simultaneous assignment. Example 3.5 can be rewritten as

```
if a = 0 then d := b
else a, b := b mod a, a
```

A question arises what happens if the components perform contradictory actions in parallel, for example,

```
do in-parallel
   x := 7
   x := 11
```

The ASM breaks down in such a case. One can argue that there are better solutions for such situations that guarantee that sequential ASMs do not break down. In the case of the program above, for example, one of the two values, 7 or 11, can be chosen in one way or another and assigned to x. Note, however, that some sequential algorithms do break down. That is a part of their behavior. If sequential ASMs do not ever break down, then no sequential ASM can be behaviorally equivalent to a sequential algorithm that does break down.

In the Euclidean algorithm, all dynamic functions are nullary. Here is a version of the algorithm where some of dynamic functions are unary. Initially $\mathit{mode} = s = 0$.

Example 3.6 (Euclidean Algorithm 2).

```
if mode = 0 then a(s), b(s), mode := Input1(s), Input2(s), 1
elseif mode = 1 then
   if a(s) = 0 then d(s), s, mode := b(s), s+1, 0
   else a(s), b(s) := b(s) mod a(s), a(s)
```

Theorem 3.7 (Sequential Characterization Theorem). *Every sequential ASM is a sequential algorithm, and every sequential algorithm is behaviorally equivalent to a sequential ASM.*

We turn our attention to parallel algorithms and quote from [4]: “The term ‘parallel algorithm’ is used for a number of different notions in the literature. We have in mind sequential-time algorithms that can exhibit unbounded parallelism but only bounded sequentiality within a single step. Bounded sequentiality means that there is an *a priori* bound on the lengths of sequences of events within any one step of the algorithm that must occur in a specified order. To distinguish this notion of parallel algorithms, we call such parallel algorithms *wide-step*. Intuitively the width is the amount of parallelism. The 'step' in 'wide-step' alludes to sequential time." Taking into account the bounded sequentiality of wide-step algorithms, they could be called "wide and shallow step algorithms".

4 Interaction

4.1 Inter-Step Interaction

One may have the impression that the algorithms of the previous section do not interact at all with the environment during the computation. This is not necessarily so. They do not interact with the environment during a step; we call such algorithm *isolated step algorithms*. But the environment can intervene between the steps of an algorithm. The environment preserves the vocabulary of the state but otherwise it can change the state in any way. It makes no difference in the proofs of the two characterization theorems whether inter-step interaction with the environment is or is not permitted.

In particular, Euclidean Algorithm 2 could be naturally inter-step interactive; the functions Input1 and Input2 do not have to be given ahead of time. Think of a machine that repeatedly applies the Euclidean algorithm and keeps track of the number s of the current session. At the beginning of session s, the user provides numbers Input1(s) and Input2(s), so that the functions Input1(s) and Input2(s) are *external*. The inter-step interactive character of the algorithm becomes obvious if we make the functions Input1, Input2 nullary.

Example 4.1 (Euclidean Algorithm 3).

```
if mode = 0 then a(s), b(s), mode := Input1, Input2, 1
elseif mode = 1 then
   if a(s) = 0 then d(s), s, mode := b(s), s+1, 0
   else a(s), b(s) := b(s) mod a(s), a(s)
```

4.2 Intra-Step Interaction

In applications, however, much of the interaction of an algorithm with its environment is intra-step. Consider for example an assignment

```
   x := g(f(7))
```

where $f(7)$ is a remote procedure call whose result is used to form another remote procedure call. It is natural to view the assignment being done within one step. Of course, we can break the assignment into several steps so that interaction is inter-step but this forces us to a lower abstraction level. Another justification of intra-step interaction is related to parallelism.

Example 4.2. This example reflects a real-world AsmL experience. To paint a picture, an AsmL application calls an outside paint applications. A paint agent is created, examines the picture and repeatedly calls the algorithm back: what color for such and such detail? The AsmL application can make two or more such paint calls in parallel. It is natural to view parallel conversations with paint agents happening intra-step.

Proviso 4.3. In the rest of this article, we concentrate on intra-step interaction and ignore inter-step interaction. By default, interaction is intra-step interaction.

4.3 The Ubiquity of Interaction

Intra-step interaction is ubiquitous. Here are some examples.

- Remote procedure calls.
- Doing the following as a part of expression evaluation: getting input, receiving a message, printing output, sending a message, using an oracle.
- Making nondeterministic choices among two or more alternatives.
- Creating new objects in the object-oriented and other paradigms.

The last two items require explanation. First we address nondeterministic choices. Recall that we do not consider distributed algorithms here. A sequential-step algorithm just follows instructions and cannot nondeterministically choose all by itself. But it can solicit help from the environment, and the environment may be able to make a choice for the algorithm. For example, to evaluate an expression

`any` x `|` x `in` $\{0, 1, 2, 3, 4, 5\}$ `where` $x > 1$

an AsmL program computes the set $\{2, 3, 4, 5\}$ and then uses an outside pseudo-random number generator to choose an element of that set. Of course an implementation of a nondeterministic algorithm may incorporate a choosing mechanism, so that there is no choice on the level of the implementation.

Re new object creation. An object-oriented program does not have the means necessary to create a new object all by itself: to allocate a portion of the memory and format it appropriately. A create-a-new-object command solicits outside help. This phenomenon is not restricted to the object-oriented paradigm. We give a non-object-oriented example. Consider an ASM rule

```
import v
   NewLeaf := v
```

that creates a new leaf say of a tree. The import command is really a query to the environment. In the ASM paradigm, a state comes with an infinite set of so-called reserve elements. The environment chooses such a reserve elements and returns it as a reply to the query.

4.4 Interaction Mechanisms

One popular interaction form is exemplified by the Remote Procedure Call (RPC) mechanism. One can think of a remote procedure call as a query to the environment where the caller waits for a reply to its query in order to complete a step and continue the computation. This interaction form is often called synchronous or blocking. Another popular interaction form is message passing. After sending a message, the sender proceeds with its computation; this interaction form is often called asynchronous or nonblocking. The synchronous/asynchronous and blocking/nonblocking terminologies may create an impression that every atomic intra-step interaction is in one of the two form. This is not the case. There is a spectrum of possible interaction forms. For example, a query may require two replies: first an acknowledgment and then an informative reply. One can think of queries with three, four or arbitrarily many replies.

Nevertheless, according to [5], there a universal form of atomic intra-step interaction: not-necessarily-blocking single-reply queries. In the previous paragraph, we have already represented a remote procedure call as a query. Sending a message can be thought of as a query that gets an immediate automatic reply, an acknowledgment that the query has been issued. Producing an output is similar. In fact, from the point of view of an algorithm issuing queries, there is no principal difference between sending a message and producing an output; in a particular application of course messages and outputs may have distinct formats.

What about two-reply queries mentioned above? It takes two single-reply queries to get two answers. Consider an algorithm A issuing a two-reply query q and think of q as a single-reply query. When the acknowledgment comes back, A goes to a mode where it expects an informative answer to q. This expectation can be seen as implicitly issuing a new query q'. The informative reply ostensibly to q is a usual reply to q'. In a similar way, one can explain receiving a message. It may seem that the incoming message is not provoked by any query. What query is it a reply to? An implicit query. That implicit query manifests itself in A's readiness to accept the incoming message. Here is an analogy. You sleep and then wake up because of the alarm clock buzz. Have you been expecting the buzz? In a way you were, in an implicit sort of way. Imagine that, instead of producing a buzz, the alarm clock quietly produces a sign "Wake up!" This will not have the desired effect, would it?

In general we do not assume that the query issuer has to wait for a reply to a query in order to resume its computation. More about that in § 6.

What are potential queries precisely? This question is discussed at length in [5]. It is presumed that potential answers to a query are elements of the state of the algorithm that issued the query, so that an answer makes sense to the algorithm.

5 Ordinary Interactive Small-Step Algorithms

Proviso 5.1. To simplify the exposition, in the rest of the paper we speak primarily about small-step algorithms. By default, algorithms are small-step algorithms.

Informally speaking, an interactive algorithm is *ordinary* if it has the following two properties.

- The algorithm cannot successfully complete a step while there is an unanswered query from that step.
- The only information that the algorithm receives from the environment during a step consists of the replies to the queries issued during the step.

Ordinary interactive algorithms are axiomatized in [5]. Some postulates of [5] refactor those of [15]. One of the new postulates is this:

Postulate 5.2 (Interaction Postulate). *An interactive algorithm determines, for each state X, a* causality relation $\vdash_X$ *between finite answer functions and potential queries.*

Here an answer function is a function from potential queries to potential replies. An answer function α is *closed* under a causality relation $\vdash_X$ if every query caused by α or by a subfunction of α is already in the domain of α. Minimal answer functions closed under $\vdash_X$ are *contexts* at X.

As before, behaviorally equivalent algorithms do the same thing in all circumstances. To make this precise, we need a couple of additional definitions. Given a causality relation $\vdash_X$ and an answer function α, define an *α-trace* to be a sequence $\langle q_1, \ldots, q_n \rangle$ of potential queries such that each q_i is caused by the restriction α_i of α to $\{q_j : j < k\}$ or by some subfunction of α_i. A potential query q is *reachable* from α under $\vdash_X$ if it occurs in some α-trace. Two causality relations are *equivalent* if, for every answer function α, they make the same potential queries reachable from α.

Definition 5.3. Two ordinary interactive algorithms are *behaviorally equivalent* if

- they have the same states and initial states,
- for every state, they have equivalent causality relations, and
- for every state and context, they both fail or they both succeed and produce the same next state. $\square$

We turn our attention to ordinary abstract state machines. Again, a machine is given by a program, a collection of states and a subcollection of initial states. We need only to describe programs.

The syntax of ordinary ASM programs is nearly the same as that of isolated state algorithms, the algorithms of [15]. The crucial difference is in the semantics of external functions. In the case of isolated step algorithms, an invocation of an external function is treated as a usual state-location lookup; see Euclidean Algorithm 2 or 3 in this connection. In the case of interactive algorithms, an invocation of an external function is a query.

The new interpretation of external functions gives rise to a problem. Suppose that you have two distinct invocations $f(3)$ of an external function $f()$ in your program. Should the replies be necessarily the same? In the case of an isolated-step program, the answer is yes. Indeed, the whole program describes one step of an algorithm, and the state does not change during the step. Two distinct lookups of $f(3)$ will give you the same result. In the case of an interactive program, the replies don't have to be the same. Consider

Example 5.4 (Euclidean Algorithm 4).

```
if mode = 0 then a, b, mode := Input, Input, 1
elseif mode = 1 then
   if a = 0 then d, mode := b, 0
   else a, b := b mod a, a
```

The two invocations of Input are different queries that may have different results. Furthermore, in the object-oriented paradigm, two distinct invocations of the same create-a-new-object command with the same parameters necessarily result in two distinct objects. We use a mechanism of template assignment to solve the problem in question [6, 7].

The study of ordinary interactive algorithms in [5, 6, 7] culminates in

Theorem 5.5 (Ordinary Interactive Characterization Theorem). *Every ordinary interactive ASM is an ordinary interactive algorithm, and every ordinary interactive algorithm is behaviorally equivalent to an ordinary interactive ASM.*

6 General Interactive Algorithms

Call an interactive algorithm *patient* if it cannot finish a step without having the replies to all queries issued during the step. While ordinary interactive algorithms are patient, this does not apply to all interactive algorithms. The algorithm

Example 6.1 (Impatience).

```
do in parallel
   if $\alpha$ or $\beta$ then x:=1
   if $\neg\alpha$ and $\neg\beta$ then x:=2
```

issues two Boolean queries α and β. If one of the queries returns "true" while the other query is unanswered, then the other query can be aborted.

Call an interactive algorithm *time insensitive* if the only information that it receives from the environment during a step consists of the replies to the queries issued during the step. Ordinary algorithms are time insensitive. Since our algorithms interact with the environment only by means of queries, it is not immediately obvious what information the algorithm can get from the environment in addition to the replies. For example, time stamps, reflecting the times when the replies were issued, can be considered to be parts of the replies.

The additional information is the order in which the replies come in. Consider for example an automated financial broker with a block of shares to sell and two clients bidding for the block of shares. If the bid of client 1 reaches the broker first, then the broker sells the shares to client 1, even if client 2 happened to issue a bid a tad earlier.

An algorithm can be impatient and time sensitive at the same time. Consider for example a one-step algorithm that issues two queries, q_1 and q_2, and then does the following. If q_i is answered while q_{2-i} is not, then it sets x to i and aborts q_{2-i}. And if the queries are answered at the same time, then it sets x to 0.

The following key observation allowed us to axiomatize general interactive algorithms. Behind any sequential-step algorithm there is a single executor of the algorithm. In particular, it is the executor who gets query replies from the environment, in batches, one after another. It follows that the replies are linearly preordered according to the time or arrival. In [8], we successfully execute the algorithm explication procedure of § 2 in the case of general interactive algorithms.

Theorem 6.2 (Interactive Characterization Theorem). *Every interactive ASM is an interactive algorithm, and every interactive algorithm is behaviorally equivalent to an interactive ASM.*

A variant of this theorem is proved in [9]. The twist is that, instead of interactive algorithms, we speak about their components there.

Patient (but possibly time sensitive) interactive algorithms as well as time insensitive (but possibly impatient) interactive algorithms are characterized in [10].

These variants of the interactive characterization theorem as well as the theorem itself are about small-step algorithms. The interactive characterization theorem is generalized to wide-step algorithms in [11].

7 Finale

The behavioral theory of small-isolated-step algorithms [15] was an after-the-fact explanation of what those algorithms were. Small-isolated-step algorithms had been studied for a long time.

The behavioral theory of wide-isolated-step algorithms was developed in [3]. Wide-isolated-step algorithms had been studied primarily in computational complexity where a number of wide-isolated-step computation models had been known. But the class of wide-isolated-step algorithms of [3] is wider. The theory was used to develop a number of tools [1], most notably the specification language AsmL [2]. Because of the practical considerations of industrial environment, intra-step interaction plays a considerable role in AsmL. That helped us to realize the importance and indeed inevitability of intra-step interaction.

The behavioral theory of intra-step interactive algorithms is developed in [5]–[11]. While intra-step interaction is ubiquitous, it has been studied very little if at all. We hope that the research described above will put intra-step interaction on the map and will give rise to further advances in specification and high-level programming of interactive algorithms.

References

[1] ASM Michigan Webpage, `http://www.eecs.umich.edu/gasm/`, maintained by James K. Huggins.

[2] The AsmL webpage, `http://research.microsoft.com/foundations/AsmL/`.

[3] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," *ACM Trans. on Computational Logic*, 4:4 (2003), 578–651.

[4] Andreas Blass and Yuri Gurevich, "Algorithms: A Quest for Absolute Definitions," Bull. Euro. Assoc. for Theor. Computer Science Number 81, October 2003, pages 195–225. Reprinted in *Current Trends in Theoretical Computer Science: The Challenge of the New Century*, Vol. 2, eds. G. Paun et al., World Scientific, 2004, 283–312.

[5] Andreas Blass and Yuri Gurevich, "Ordinary Interactive Small-Step Algorithms, I", *ACM Trans. on Computational Logic*, to appear. Microsoft Research Tech. Report MSR-TR-2004-16.

[6] Andreas Blass and Yuri Gurevich, "Ordinary Interactive Small-Step Algorithms, II", *ACM Trans. on Computational Logic*, to appear. Microsoft Research Tech. Report MSR-TR-2004-88.

[7] Andreas Blass and Yuri Gurevich, "Ordinary Interactive Small-Step Algorithms, III", *ACM Trans. on Computational Logic*, to appear. Microsoft Research Tech. Report MSR-TR-2004-88, mentioned above, covers this article as well; the material was split into two pieces by the journal because of its article-length restriction.

[8] Andreas Blass, Yuri Gurevich, Dean Rosenzweig and Benjamin Rossman, "General Interactive Small-Step Algorithms", in preparation.

[9] Andreas Blass, Yuri Gurevich, Dean Rosenzweig and Benjamin Rossman, "Composite Interactive Algorithms" (tentative title), in preparation.

[10] Andreas Blass, Yuri Gurevich, Dean Rosenzweig and Benjamin Rossman, "Interactive Algorithms: Impatience and Time Sensitivity" (tentative title), in preparation.

[11] Andreas Blass, Yuri Gurevich, Dean Rosenzweig and Benjamin Rossman, "Interactive Wide-Step Algorithms", in preparation.

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

[13] Yuri Gurevich, "Evolving Algebras: An Introductory Tutorial", Bull. Euro. Assoc. for Theor. Computer Science 43, February 1991, 264–284. A slightly revised version is published in *Current Trends in Theoretical Computer Science*, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266-292.

[14] Yuri Gurevich, "Evolving Algebra 1993: Lipari Guide," in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 9–36.

[15] Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms," *ACM Trans. on Computational Logic* 1:1 (2000), 77–111.

[16] Andrey N. Kolmogorov, "On the Concept of Algorithm", *Uspekhi Mat. Nauk* 8:4 (1953), 175–176, Russian.

[17] Andrey A. Markov, "Theory of Algorithms", Transactions of the Steklov Institute of Mathematics, vol. 42 (1954), Russian. Translated to English by the Israel Program for Scientific Translations, Jerusalem, 1962.