Report Number: WUCS-91-42

1991-07-01

The Spectrum Simulation System: A Formal Approach to Distributed Algorithm Development Tools

Kenneth J. Goldman

We present the Spectrum Simulation System, a new research tool for the design and study of distributed algorithms. Based on the formal Input/Output Automation model of Lynch and Tuttle, Spectrum allows one to express distributed algorithms as collections of I/O automata and simulate them directly in terms of the semantics of that model. This permits integration of algorithm specification, design, debugging, analysis, and proof of correctness within a single formal framework that is natural for describing distributed algorithms. Spectrum provides a language for expressing algorithms as I/O automata, a simulator for generating algorithm executions, and a graphics interface for... **Read complete abstract on page 2.**

The Spectrum Simulation System: A Formal Approach to Distributed Algorithm Development Tools

Kenneth J. Goldman

Complete Abstract:

We present the Spectrum Simulation System, a new research tool for the design and study of distributed algorithms. Based on the formal Input/Output Automation model of Lynch and Tuttle, Spectrum allows one to express distributed algorithms as collections of I/O automata and simulate them directly in terms of the semantics of that model. This permits integration of algorithm specification, design, debugging, analysis, and proof of correctness within a single formal framework that is natural for describing distributed algorithms. Spectrum provides a language for expressing algorithms as I/O automata, a simulator for generating algorithm executions, and a graphics interface for constructing systems of automata and observing their executions. We show that the properties of the I/O automation model provide a solid foundation for algorithm development tools. For example, using I/O automation composition, Spectrum users may define composed types hierarchically, study simulations at varying levels of detail, and create specialized debugging and analysis devices. These devices, called spectators, are written in the Spectrum language just as any other system component, and can monitor algorithm executions for correctness and performance without interfering with the algorithm. The system is designed to support experimentation with algorithm. For example, the system separates algorithms from the system configurations in which they are to run, allowing users to vary them independently. Also the message system may be modeled explicitly as as automation, permitting users to study algorithms under different communications assumptions simply by substituting one automation for another.

The Spectrum Simulation System: A Formal Approach to Distributed Algorithm Development Tools

Kenneth J. Goldman

WUCS-91-42

July 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

The Spectrum Simulation System: A Formal Approach to Distributed Algorithm Development Tools

Kenneth J. Goldman [1]
Department of Computer Science
Washington University
St. Louis, Missouri 63130–4899
kjg@cs.wustl.edu

[1]Most of this work was conducted at the Massachusetts Institute of Technology Laboratory for Computer Science and was supported in part by the National Science Foundation under Grant CCR-86-11442, by the Office of Naval Research under Contract N00014-85-K-0168, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, and by an Office of Naval Research graduate fellowship.

Abstract

We present the Spectrum Simulation System, a new research tool for the design and study of distributed algorithms. Based on the formal Input/Output Automaton model of Lynch and Tuttle, Spectrum allows one to express distributed algorithms as collections of I/O automata and simulate them directly in terms of the semantics of that model. This permits integration of algorithm specification, design, debugging, analysis, and proof of correctness within a single formal framework that is natural for describing distributed algorithms. Spectrum provides a language for expressing algorithms as I/O automata, a simulator for generating algorithm executions, and a graphics interface for constructing systems of automata and observing their executions.

We show that the properties of the I/O automaton model provide a solid foundation for algorithm development tools. For example, using I/O automaton *composition*, Spectrum users may define composed types hierarchically, study simulations at varying levels of detail, and create specialized debugging and analysis devices. These devices, called *spectators*, are written in the Spectrum language just as any other system component, and can monitor algorithm executions for correctness and performance without interfering with the algorithm.

The system is designed to support experimentation with algorithms. For example, the system separates algorithms from the system configurations in which they are to run, allowing users to vary them independently. Also, the message system may be modeled explicitly as an automaton, permitting users to study algorithms under different communications assumptions simply by substituting one automaton for another.

Keywords: distributed computing, distributed algorithms, software development tools, specification, input/output automata

1 Introduction

With continued advances in computer communications technology and the resulting proliferation of distributed computing systems, methodologies and tools supporting the design of distributed algorithms will become increasingly important. This is not only because more people will be designing distributed algorithms, but also because the algorithms themselves are likely to become more specialized and complex, tailored to particular distributed applications. And unlike sequential algorithms, distributed algorithms must cope with arbitrary communication delays and both processor and communication failures. The fact that communication delays are unpredictable means that distributed algorithms must also cope with arbitrary interleaving of processor steps. Therefore, a given program's computation may unfold nondeterministically. This makes designing and reasoning about distributed algorithms inherently difficult. Consequently, researchers have turned to formal models of distributed systems in order to reason about their algorithms. For example, the I/O automaton model of Lynch and Tuttle [26, 27] is particularly well suited for the study of distributed algorithms; it allows one to state natural correctness conditions, give precise algorithm descriptions, and construct careful correctness proofs.

We claim that formal models are important not only as a means to analyzing and proving the correctness of distributed algorithms, but also as the basis of software tools for designing better algorithms. The aim of this paper is to demonstrate how distributed algorithm specification, design, debugging, analysis, and proof of correctness may be integrated within a single formal framework that is natural for describing a wide range of distributed algorithms. Such integration not only saves one from translation between different models and languages, but also allows facts discovered during simulation and debugging to be more easily incorporated into the correctness proof, and allows properties used in the proof to be checked mechanically during simulation.

We present the Spectrum Simulation System, a new research tool for the design and study of distributed algorithms. Spectrum consists of a programming language and simulator based on the I/O automaton model. Users express distributed algorithms as collections of I/O automata and simulate them directly, in a way that is faithful to the semantics of the formal model. A graphical user interface is provided for constructing systems of automata and visualizing their executions. In presenting the system, we describe how the salient features of the I/O automaton model provide a solid foundation for distributed algorithm development tools. For example, using I/O automaton *composition*, Spectrum users may define composed types hierarchically, study simulations at varying levels of detail, and create specialized debugging and analysis devices.

In the remainder of this introduction, we define the scope of the problem by clarifying what we mean by a distributed algorithm and outlining those aspects of distributed computation that we consider important to capture within a distributed algorithm development tool. We conclude this discussion by describing the design philosophy behind Spectrum and listing a number of specific design goals.

1.1 Preliminaries

A *distributed system* consists of a collection of geographically separated computers linked together by a *network*. In general, the *network topology*, the arrangement of communication links between processors, may be arbitrary. *Processes*, program threads running on the computers, may communicate with each other by sending *messages* over the network, but do not have any other means of communication, such as a shared memory. Processes are autonomous, meaning that they determine when to send messages to other processes. That is, a process cannot prevent another process from sending a message. Processes do not have synchronized clocks, and their instruction execution rates may differ. This implies that processes are *asynchronous*; their steps may be arbitrarily interleaved. Network communication is also asynchronous, meaning that the acts of sending and receiving a message are separated (often arbitrarily) in time.

Like any computer system, distributed systems are prone to *failures*. However, unlike centralized systems, portions of a distributed system may continue to be useful while other portions are "down." We classify the types of failures that may occur in a distributed system into *process failures* and *communication failures*. Process failures range from simple stopping faults (crashes) to malicious faults, in which faulty processors attempt to corrupt the computation of the rest of the system by sending incorrect or conflicting messages. If we assume that a communication link is supposed to deliver each message exactly once and in the order sent, then communication failures may involve losing messages, reordering messages, delivering messages that were never sent, or delivering messages multiple times.

In order to coordinate the activities of processes in a distributed system, it is necessary to have agreed-upon protocols. These protocols are called *distributed algorithms*. Typical problems solved by distributed algorithms include leader election, distributed consensus, mutual exclusion, and global snapshot. Distributed algorithms are usually designed with particular assumptions about the underlying system in mind. For example, one might assume that the only process failures are crash failures and that all messages sent eventually arrive. In addition to failure assumptions, one might make assumptions about the network topology or about the existence of unique process identifiers. The set of assumptions makes a great impact on the algorithmic solution. It is often interesting to consider what happens when an algorithm's assumptions are violated. For example, consider a distributed algorithm designed with the assumption that messages are delivered in order. Depending on the particular algorithm, delivering messages out of order might cause the algorithm to produce an incorrect result, might have no effect whatsoever, or might cause the algorithm to produce a correct result but with degraded (or superior!) performance.

The particular combination of process and communication failures that an algorithm must tolerate forms part of the *problem specification*. A specification is usually presented in terms of the input/output relationship between the algorithm and its *environment*. Since an algorithm has no control over its environment, a problem specification usually says that an algorithm will satisfy certain *safety* and *liveness* conditions, provided that its inputs are *well-formed*. The safety conditions essentially say that the algorithm *never* does anything "wrong," and the liveness (progress)

properties essentially say that the algorithm *eventually* does something "right." An algorithm is said to be *correct* if it satisfies both the safety and liveness conditions (whenever its inputs from the environment are well-formed).

We need formal models to help overcome the inherent difficulty of designing distributed algorithms that stems from the arbitrary interleaving of process steps. Informal arguments and software testing are inadequate substitutes for formal methods, since anything short of a complete proof is likely to miss "bad" executions — executions in which the particular choice of process step interleaving leads to the violation of safety or liveness requirements. Formal models, such as the I/O automaton model, are useful for stating problem specifications, describing algorithms precisely, and constructing careful proofs of correctness. However, formal proofs of correctness are often long, hard, and tedious. If an algorithm is incorrect, much effort can be wasted in attempting to prove its correctness. Testing can help one to discover many errors in algorithms quickly and easily, before delving into a correctness proof. Furthermore, simply constructing a correctness proof for an algorithm may not reveal enough intuition into how the algorithm works in order to lead to improvements in the algorithm. For these reasons, it is important to have research tools for *simulating* distributed algorithms.

Simulation allows one to test and debug algorithms, and can reveal intuition that is helpful in understanding algorithms and constructing correctness proofs for them. In conjunction with appropriate graphical visualization techniques, simulation facilitates the study of algorithm performance under varying conditions, something not easily done in the context of a proof. But because successful testing alone is not sufficient cause to believe that an algorithm is correct, one must still construct a correctness proof as part of the algorithm development cycle. Therefore, it is important that the semantics of the simulation language be consistent with the formal model in which the proof is to be constructed. In addition, using an appropriate formal model as the basis of a simulation tool leaves open the possibility of integrating the entire algorithm development process: specification, design, debugging, analysis, and proof of correctness.

1.2 Design Philosophy

Motivated by the ideas we have just described, the following design principles were used to guide development of the Spectrum Simulation System.

The design must be faithful to a formal model. Since our aim is to integrate theoretical modelling techniques with algorithm simulation, the simulation language and its semantics (as well as the implementation of the simulator) must remain faithful to the formal model. Any departure from the formal model jeopardizes effective integration of the two. For example, it is only possible to mechanically check executions of an algorithm against properties stated in the proof if the semantics of the simulation match the semantics of the model. By remaining faithful to a theoretical model, we also benefit from having a well-defined semantics on which to base the language and implementation.

The language should be natural for expressing a large class of distributed algorithms. Of course, this means we must choose a sufficiently simple model so that the resulting language mechanisms encourage writing straightforward algorithm descriptions, and so that the resulting simulations are easily comprehended. But in addition, we want the model and language to reflect the fact that processes in distributed systems generate output autonomously and may receive input at arbitrary times. Also, since distributed algorithms can be designed with many different communication assumptions, the system should provide support for varying these assumptions. Therefore, one should be able to model communication mechanisms explicitly. Many distributed algorithms make use of unbounded state, such as message counters or history information. In order not to rule out such algorithms, we require that the language allow processes to have infinite state sets (in principle). Finally, we require that the language have built-in data types and control flow mechanisms that are convenient for describing distributed algorithms.

The simulation tool should facilitate the discovery of new insights about an algorithm through experimentation and exploration. In general, support for experimentation means that it should be easy to modify the algorithm and manipulate the simulation. User effort should be focused on experimenting with *algorithms* rather than finding obscure program errors. In addition, the language should provide mechanisms for modularity, so that algorithm components may be studied individually or replaced with other components. This modularity should have a hierarchical structure, so that simulations can be studied at different levels of detail. In addition, the system should support writing user-defined debugging and analysis tools as separate modules, so as not to clutter up the algorithm or interfere with its execution. In fact, all logically independent concerns should be orthogonal in the language. For example, it should be possible to modify each of the following aspects of a simulation independently: the algorithm being studied, the system configuration, the control of visualization, and the mechanisms for debugging and analysis. And, of course, flexible mechanisms should be provided for controlling and studying executions. Examples of such mechanisms include automatic detection of invariant violations, simple graphical mechanisms for configuring systems and controlling visualization, a choice of scheduling options, and the ability to go back to an earlier point in a simulated execution.

The design should achieve economy and integration. In general, a system is easier to build, learn, and use when a small set of tools provide all the necessary functionality. Therefore, it is desirable to use the same language mechanisms for writing programs, creating debugging tools, specifying invariants, and setting up visualization. In addition, a single graphical interface should be used for both constructing the system configuration and controlling the simulation.

The above design principles summarize the design philosophy for the Spectrum Simulation System, and their influence is evident in the Spectrum design.

The remainder of the paper is organized as follows. In Section 2, we review the I/O automaton model, the theoretical foundation for our work. This is followed by the presentation of the Spectrum Simulation System. The tool consists of three main components, the *programming language*, the *simulator*, and the *user interface*. Central to the design of Spectrum is a clear separation of *automaton types*, which are the different kinds of components in an automaton system, and the

configuration, which defines the number of instances of each of those types and the relationships among them. The programming language, defined in Section 3, provides constructs for describing distributed algorithms as I/O automaton types. The language provides constructs that support algorithm visualization and mechanical checking of state invariants. I/O automaton types are separately instantiated in order to form an automaton system configuration. The Spectrum simulator, described in Section 4, provides facilities for generating executions of these automaton systems. The graphical user interface, described in Section 5, is used both for defining the configuration and for controlling the simulation. Spectrum is written entirely in C [17] and runs on DEC Microvax workstations. The user interface is built on top of the X11 window system [32]. We conclude the paper with an evaluation of Spectrum in terms of the design goals described above, and suggest a number of directions for future research.

There are a number of other languages and systems based on formal models that support the study of concurrent algorithms. These are not all designed for simulation and visualization of distributed algorithms, but are sufficiently related that one might imagine using them (or extending them) for that purpose. We do not include a survey of related languages and systems here, but a detailed comparison of Spectrum with several of these languages and systems (including Occam [16, 30], Unity [4], Statemate [12], and DEVS [35]), may be found elsewhere[8].

2 The Model

The I/O Automaton model [26, 27] has been chosen as the foundation of the Spectrum Simulation System primarily because it is a natural model for describing distributed algorithms. Using a variety of techniques, careful correctness proofs have been constructed in this model for a wide range of distributed algorithms (for examples, see [3, 5, 10, 21, 23, 24, 25, 26, 33, 34]). In this section, we present a review of the I/O automaton model adapted from [27]. Interested readers are referred to that paper for more details, motivation, examples, and results.

2.1 I/O Automata

I/O automata are best suited for modelling systems in which the components operate asynchronously. Each system component is modeled as an I/O automaton, which is essentially a nondeterministic (possibly infinite state) automaton with an action labeling each transition. An automaton's actions are classified as either 'input', 'output', or 'internal'. An automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action. An automaton is said to be *closed* if it has no input actions; it models a closed system that does not interact with its environment.

Formally, an *action signature* S is a partition of a set $acts(S)$ of *actions* into three disjoint sets $in(S)$, $out(S)$, and $int(S)$ of *input actions*, *output actions*, and *internal actions*, respectively. We denote by $ext(S) = in(S) \cup out(S)$ the set of *external actions*. We denote by $local(S) = out(S) \cup int(S)$ the set of *locally controlled actions*. An I/O automaton A consists of five components:

- an action signature $sig(A)$,
- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*,
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$, and
- an equivalence relation $part(A)$ partitioning the set $local(A)$ into at most a countable number of equivalence classes.

The equivalence relation $part(A)$ will be used in the definition of fair computation. Each class of the partition may be thought of as a separate process. We refer to an element (s', π, s) of $steps(A)$ as a *step* of A. If (s', π, s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled.* This means that an automaton is unable to block its input.

An *execution* of A is a finite sequence $s_0, \pi_1, s_1, \ldots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i and $s_0 \in start(A)$. The *schedule* of an execution α is the subsequence of α consisting of the actions appearing in α. The *behavior* of an execution or schedule α of A is the subsequence of α consisting of *external* actions. The sets of executions, finite executions, schedules, finite schedules, behaviors, and finite behaviors are denoted $execs(A)$, $finexecs(A)$, $scheds(A)$, $finscheds(A)$, $behs(A)$, and $finbehs(A)$, respectively. The same action may occur several times in an execution or a schedule; we refer to a particular occurrence of an action as an *event.*

2.2 Composition

We can construct an automaton modelling a complex system by composing automata modelling the simpler system components. When we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π, all automata having π as an action perform π simultaneously (automata not having π as an action do nothing).

Since we require that at most one system component controls the performance of any given action, we must place some compatibility restrictions on the collections of automata that may be composed. A countable collection $\{S_i\}_{i \in I}$ of action signatures is said to be *strongly compatible* if for all $i, j \in I$ satisfying $i \neq j$ we have

1. $out(S_i) \cap out(S_j) = \emptyset$,
2. $int(S_i) \cap acts(S_j) = \emptyset$, and
3. no action is contained in infinitely many sets $acts(S_i)$.

We say that a collection of automata is *strongly compatible* if the corresponding collection of action signatures is strongly compatible.

The *composition* $S = \prod_{i \in I} S_i$ of a countable collection of strongly compatible action signatures $\{S_i\}_{i \in I}$ is defined to be the action signature with

- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$,
- $out(S) = \cup_{i \in I} out(S_i)$, and
- $int(S) = \cup_{i \in I} int(S_i)$.

The *composition* $A = \prod_{i \in I} A_i$ of a countable collection of strongly compatible automata $\{A_i\}_{i \in I}$ is the automaton defined as follows:[1]

- $sig(A) = \prod_{i \in I} sig(A_i)$,
- $states(A) = \prod_{i \in I} states(A_i)$,
- $start(A) = \prod_{i \in I} start(A_i)$,
- $steps(A)$ is the set of triples $(\vec{s_1}, \pi, \vec{s_2})$ such that, for all $i \in I$, if $\pi \in acts(A_i)$ then $(\vec{s_1}[i], \pi, \vec{s_2}[i]) \in steps(A_i)$, and if $\pi \notin acts(A_i)$ then $\vec{s_1}[i] = \vec{s_2}[i]$, and
- $part(A) = \cup_{i \in I} part(A_i)$.

Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1}\ldots$ of A, let $\alpha|A_i$ (read "α projected on A_i") be the sequence obtained by deleting $\pi_j\vec{s_j}$ when $\pi_j \notin acts(A_i)$ and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$.

2.3 Fairness

Of all the executions of an I/O automaton, we are primarily interested in the 'fair' executions — those that permit each of the automaton's primitive components (i.e., its classes or processes) to have infinitely many chances to perform output or internal actions. The definition of automaton composition says that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the essential structure of the system's primitive components. In the model, therefore, being fair to each component means being fair to each equivalence class of locally controlled actions. A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of $part$(A):

1. If α is finite, then no action of C is enabled in the final state of α.
2. If α is infinite, then either α contains infinitely many events from C, or α contains infinitely many occurrences of states in which no action of C is enabled.

[1] Here $start(A)$ and $states(A)$ are defined in terms of the ordinary Cartesian product, while $sig(A)$ is defined in terms of the composition of action signatures just defined. Also, we use the notation $\vec{s}[i]$ to denote the ith component of the state vector $\vec{s}$.

2.4 Problem Specification

A 'problem' to be solved by an I/O automaton is formalized as a set of (finite and infinite) sequences of external actions. An automaton is said to *solve* a problem P provided that its set of fair behaviors is a subset of P. Although the model does not allow an automaton to block its environment or eliminate undesirable inputs, we can formulate our problems (i.e., correctness conditions) to require that an automaton exhibits some behavior only when the environment observes certain restrictions on the production of inputs.

We want a problem specification to be an interface together with a set of behaviors. We therefore define a *schedule module* H to consist of two components, an action signature $sig(H)$, and a set $scheds(H)$ of *schedules*. Each schedule in $scheds(H)$ is a finite or infinite sequence of actions of H. Subject to the same restrictions as automata, schedule modules may be composed to form other schedule modules. The resulting signature is defined as for automata, and the schedules $scheds(H)$ is the set of sequences β of actions of H such that for every module H' in the composition, $\beta|H'$ is a schedule of H'.

It is often the case that an automaton behaves correctly only in the context of certain restrictions on its input. A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors. A set of sequences $\mathcal{P}$ is said to be *prefix-closed* if $\beta \in \mathcal{P}$ whenever both β is a prefix of α and $\alpha \in \mathcal{P}$. A module M (either an automaton or schedule module) is said to be *prefix-closed* provided that $finbehs(M)$ is prefix-closed. Let M be a prefix-closed module and let $\mathcal{P}$ be a nonempty, prefix-closed set of sequences of actions from a set Φ satisfying $\Phi \cap int(M) = \emptyset$. We say that M *preserves* $\mathcal{P}$ if $\beta\pi|\Phi \in \mathcal{P}$ whenever $\beta|\Phi \in \mathcal{P}$, $\pi \in out(M)$, and $\beta\pi|M \in finbehs(M)$. Informally, a module *preserves* a property $\mathcal{P}$ iff the module is not the first to violate $\mathcal{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathcal{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathcal{P}$. One can prove that a composition preserves a property by showing that each of the component automata preserves the property.

The I/O automaton model is only one of a number of formal models that have been used for reasoning about concurrent systems. As part of our presentation of the Spectrum programming language, we highlight those features of the I/O automaton model that make it particularly well-suited for our purposes. Readers interested in a more detailed discussion of our choice of the I/O automaton model as the basis of this work, with a discussion of several alternative models (including CSP [15], Unity [4], and Statecharts [13, 14]), may refer to [8]. A review of alternative models with an emphasis on techniques for proving algorithm correctness is contained in [26].

3 The Language

The Spectrum programming language is the first executable language based on the I/O automaton model. In the literature, the transition relations of I/O automata typically have been described using variants of the "precondition/effect" notation of Lynch and Tuttle [27] based on Dijkstra's

guarded commands. In this notation, each action has a precondition that maps each state of the automaton to a boolean value, and the action is enabled in exactly those states in which the precondition is true. (Since input actions are always enabled, their preconditions are taken to be true in all states.) Similarly, each action has an effect that defines the new state of the automaton based on the action and the state from which the action occurs. However, the notations used to express these preconditions and effects have, until now, been rather *ad hoc*. Furthermore, authors usually have resorted to prose to define the data types and initial values of state components, as well as the partition of locally controlled actions. The Spectrum programming language provides well-defined constructs for expressing each of the five basic components of an I/O automaton: the signature, states, initial states, nondeterministic transition relation, and partition of locally-controlled actions.

Traditionally, I/O automaton descriptions have treated action "arguments" as part of the action name, allowing a given automaton to have infinitely many actions. Since we cannot evaluate preconditions for infinitely many actions in finite time, Spectrum separates the traditional precondition into two parts: a PRE (precondition) clause and a SEL (selection) clause. The PRE clause determines if there exists an assignment to the arguments of the action that would result in an enabled action, and the SEL clause is used to select the particular argument values for an enabled action. This will become clear as we present the details of the language.

Linguistic support for verification, analysis, and visualization are also provided. The language provides a means to express state invariants (predicates on the automaton state) to be checked after each of its steps in an execution. We also present a mechanism called a *spectator*, a separate I/O automaton having only input actions, that is useful for mechanically verifying during simulation that an automaton's execution is among the set of executions permitted by its specification, as well as for keeping track of various properties of an execution (such as the number of times a particular component enters its critical section) for analysis purposes. In addition, a mechanism is provided for defining pseudovariables, which may be mapped to colors in a graphical display of the algorithm execution. In keeping with our design goals, all of this extra language support is provided in such a way as not to obscure the algorithm being studied. The "extra" pieces of code that are present only for purposes of studying the algorithm are clearly separated from the algorithm itself. In spite of this separation, the mechanisms themselves are well-integrated with the rest of the language: the same sorts of expressions used to describe the algorithm are also used to define invariants, spectators, and pseudovariables.

The language is interpreted, provides a convenient set of built-in data types, and is statically type-checked. These features allow algorithm modifications to be made quickly, make it easier to express algorithms at a high level, and permit users to concentrate on debugging their *algorithms*, rather than on finding obscure errors in their *programs*.

The remainder of the section is organized as follows. We begin in Section 3.1 by highlighting certain aspects of the model that form a solid foundation for an implementable programming language, as well as those that must be compromised slightly in order to achieve a practical implementation. Then, in Section 3.2 we discuss a major design decision of the Spectrum Simulation System, namely

the separation of I/O automaton types from the I/O automaton system configuration. Following this, Section 3.3 contains the details of specific language mechanisms for defining I/O automaton types. Section 3.4 contains an example of an automaton type for LeLann's leader election algorithm [19]. Finally, Section 3.5 describes special language support for verification, analysis, and visualization.

3.1 Theoretical Foundations

In the previous section, we reviewed the I/O automaton model on which the Spectrum programming language is based. Certain features of the model can be used directly as the basis for a programming language, but there are also some features of the model that must be compromised slightly in order to produce an implementable language.

We begin with the model features that we capture directly in the language. As in the model, automaton descriptions in Spectrum consist of five components: a set of states, a set of initial states, a set of actions divided into input actions and output actions, a transition relation, and a partition of the output actions into classes.[2] And the language, by prohibiting preconditions on input actions, guarantees that these actions are always enabled. We also guarantee that output actions are under the control of only one automaton. In this way, Spectrum we can describe systems of autonomous components. And the semantics of a state transition in the language is consistent with the model: every state transitions in each automaton's transition relation is treated as an atomic step. Therefore, since shared actions take place simultaneously at all the participating automata, it is easy to reason about communication among the automata. We also permit automata to be composed to form complex systems. This gives rise to flexible modularity and straightforward encapsulation mechanisms. System modules can be composed in a variety of ways, and reasoning about modules is accomplished in terms of the actions that occur at their boundaries. Although the above is not a complete list of the features of the language, it should be clear that most of the features of the model are suitable for an implementable programming language.

However, there are several aspects of the model that cannot be implemented directly. Obviously, it is impossible to implement a truly infinite-state automaton on a digital computer with finite storage. Nonetheless, we can support data structures that give the illusion of an infinite state set and allow the state space of automata to be quite large — large enough for any practical algorithm. Similarly, the model permits systems to have infinitely many automata, but our implementation supports only a finite number.[3]

Since the implementation must run on a deterministic machine, we provide randomization in the language and the scheduler in place of nondeterministic choice in the model. Fortunately, the model does not rely on nondeterminism in the automata-theoretic sense. That is, nondeterminism is not used in the model to "guess" a correct solution to be deterministically checked. Rather, nondeterminism is merely present to permit a large number of possible executions of the algorithm.

[2]For simplicity, we treat internal actions as output actions that are not inputs to other components.

[3]We are considering the addition of language constructs for for creating new automata and classes dynamically to provide the illusion of an infinite set.

All of these executions are supposed to be correct, as opposed to the usual automata-theoretic case where only one need be correct. This kind of nondeterminism is useful for program development. One can first write a loosely structured program, prove it correct using the model, and then "tune" it for performance by placing additional preconditions on locally controlled actions with the knowledge that safety properties still hold. Since the intention is to build a tool that encourages experimentation with algorithms, and since generating many different executions of an algorithm is one way to achieve this, we provide randomization in the language rather than requiring that algorithm designers transform their general solutions to ones that make fixed deterministic choices.

The notion of fairness is important in both the model and the implementation, but is only secondarily a language consideration. One might imagine various algorithms for scheduling automaton classes, which may or may not satisfy the fairness requirement imposed by the model. However, the only requirement for the *language* is that one be able to specify the set of classes to which one must be fair. Fairness, then, becomes the responsibility of the scheduler.

3.2 A Separation of Concerns: Automaton Types vs. Configurations

The purpose of the Spectrum programming language is to express *I/O automaton types*, the building blocks of *I/O automaton systems*. An automaton type defines the signature, states, transition relation, and action partition of potentially many different automata. Having defined a collection of automaton types in the language, one separately supplies a *configuration* that defines the set of *instances* of automaton types to be simulated. In other words, the language is not used to define the entire system to be simulated, but only to define the different kinds of automata that may exist in a system. This division is central to the design of the Spectrum Simulation System. It separates the algorithm description from the system configuration in which the algorithm is to run, and allows one to experiment with algorithms by varying the system configuration independently.

A configuration defines the automaton instances and relationships between them. Every configuration specifies the automaton type of each instance, and includes a unique automaton id for each instance (assigned automatically by the system). In addition, one may assign each instance a name, and may specify a set of directed edges that organizes the instances into an arbitrary graph. And one may create new automaton types by composing other automaton types; each instance of such a type is then a composition of several instances of other types. If an instance's type is such a "composed type," then we say that it is the "parent" of each of its instantiated components; this hierarchical relationship is also included as part of the configuration data. One purpose of the configuration is to break symmetry. It may be used in arbitrary ways to define the signatures and transition relations of automata. For example, in a configuration of several instances arranged in a ring, one might use directed edges between instances to specify which of the instances are neighbors.

It is helpful to think of an automaton type as a program and an automaton instance as a single invocation of that program. Each instance of a given automaton type has the same program, but that program may reference information present in the configuration. Thus, two instances of

the same automaton type may have different initial states, signatures, transition relations, and partitions. Because of this, we say that an automaton type is *parameterized* by the configuration.

3.3 Language Constructs

In this section, we present the Spectrum programming language constructs for defining the states, initial states, signatures, transition relations, and partitions for I/O automaton types. The mechanisms for defining an I/O automaton system configuration are presented in Section 5.

3.3.1 Data Types

The Spectrum programming language is strongly typed and statically type-checkable in order to save users from wasting time searching for obscure errors in their code. The state components of an automaton, the arguments of actions, the parameters of classes (to be explained later), and the values of expressions in the language all have associated data types that are checked for compatibility when an automaton types file is loaded into the simulator.

There are four *base types*: integers, booleans, automaton id's, and strings. In addition, the language provides five *type constructors* for building more complicated structures from the base types: tuples with named fields, sets, multisets, sequences, and mappings, each with the expected set of operators. Constructed types may be arbitrarily nested and assigned mnemonic names. For example, the line

```
DATA buffer multiset(tuple(msg:string,dest:automaton_id))
```

defines the type *buffer* to be a multiset of tuples, where each tuple has a string and an automaton_id, with field names *msg* and *dest*, respectively. Our notion of type equality is structural equality, where names of tuple fields are considered to be part of the structure of a type. One may think of a DATA declaration as a macro definition. Recursive type declarations are not permitted. In addition to its own type-specific operations, every data type, including all constructed types, has operations for comparison and assignment. Data types for the return values of type-specific operations are inferred (statically) from the data types of the arguments. In order to prevent references to undefined variables, each base type in the language has a default value, which is assigned to that variable by the system when no initial value is specified by the programmer. Similarly, each type constructor has a default value, defined recursively on the basis of the defaults for its component types.

Automaton id's are special in that their operations provide the ability to access configuration data. For example, if `x` is an automaton id, then `neighbors(x)` returns the set of automaton id's for the neighbors (in the configuration graph) of the automaton instance with id `x`. The function `self()` returns the id of the automaton calling the function. An automaton id may also be used to reference the following information about the associated automaton instance in a configuration: its name (a user-supplied string), its parent in the composition hierarchy, its neighbors adjacent to incoming edges in the graph, and its neighbors adjacent to outgoing edges in the graph. An

operation is also provided for obtaining the set of automaton id's for all instances of a given automaton type in the configuration.

A complete description of the data types and their associated operations is provided in [8]. Currently, there is no provision in the language for writing user-defined operations. However, the language implementation is such that new operations may be added easily. Note that (with the exception of record notation) all of our syntax uses an applicative style; there are no infix operators. This adds to the length of our programs and can impair readability somewhat, but is not an inherent problem with the language. One might imagine "sugared" versions of the syntax that could be preprocessed into our applicative style.

3.3.2 Action Types

Action types define the different kinds of actions that may be used in the signatures of automaton types. Since a given action may be shared by many different automaton types, action types are declared outside of the scope of any automaton type definition. Each action is declared with a *name* and an *argument type*. The name is used to identify the action in the signatures of automata and in executions. The argument type defines the data type for the argument of the action. For example,

```
ACTION send tuple(msg:string, to:automaton-id)
```

declares an action with the name "send" and an argument that is a tuple consisting of a text string (named *msg*) and an automaton-id (named *to*). In descriptions of the transition relations of automaton types, the argument of an action is referenced by the name `a`, and record notation is used to refer to the argument components. For example, in the context of an event for the action above, one would refer to the first component of the argument by `a.msg`.

In addition to the user-defined argument for each action type, there is an implicit argument `a.owner` of type automaton-id, which names the automaton for which the action is an output. This argument may be referenced in the same way as the user-defined arguments. Having this argument ensures that each action is under the control of a single system component, as required by the I/O automaton model.

Simply declaring an action type does not associate it with any particular automata. It is the signature of an automaton that determines which are its input and output actions. When events take place during a system execution, the set of participating automata is determined according to the automaton signatures; each participant automaton takes a step according to its transition relation, its current state, the action name, and the action argument values. Next we describe how the states, signatures, and transition relations of automaton types are defined.

3.3.3 Automaton Types

There are two ways to define an automaton type. The first is to write an explicit textual definition in the Spectrum programming language. The second is to compose several automaton types to

form a new "composed" automaton type. In this section, we consider only the first method; the second method is part of the configuration process, discussed in Section 5.

States The first part of an automaton type definition, following the automaton type name, is the state declaration. The set of states for an automaton type is defined with a data type. For example, the line

```
STATE tuple(status:integer, buff:set(tuple(msg:string, to:automaton-id)))
```

says that each instance of this automaton type has two state components: an integer *status*, and a set *buff* of (string, automaton-id) pairs. The set of states for an automaton with this state definition would consist of the set of all possible assignments to these components. Thus, an automaton with this state definition would have infinitely many states. As we mentioned earlier, there are physical limitations of the computer, such as the largest representable integer or the amount of memory available for storing text strings, that prevent us from implementing a truly infinite state automaton. However, the language gives us the power to express infinite state automata that may be implemented up to the limitations of the physical architecture.

As in the I/O automaton model, an automaton's state components are private; they may be modified only by the occurrence of an action of that automaton. In the definition of a transition relation, one refers to the state as `s`, and uses record notation to refer to particular state components. For example, given the above state declaration, `s.status` would refer to the value of the first state component.

Signatures The action signature of an automaton type consists of a set of input actions and a set of output actions. Recall that action types are defined outside of the scope of any automaton type. In order to add a particular action type to the signature of an automaton type, one simply lists the action type name, indicating whether it is to be classified as an input action or an output action. For example, recall the action type `send` defined earlier. The line

```
INPUT send
```

says that all actions of type `send` are input actions to automata of this type. However, for any particular action type, we may not wish that an automaton have in its signature all the actions for all possible values of the argument. Therefore, the language provides a mechanism for restricting the argument values for each action type in the signature. Such restrictions are accomplished using a WHERE clause. For example, instead of the previous line, we might write:

```
INPUT send WHERE set_el(neighbors(self()), a.to)
```

This line specifies that the automaton being defined does not have all actions of type `send` as input actions, but only those where the "to" component of the argument is an element of the set of neighbors of the automaton in the configuration. (Recall that `a.to` refers to the "to" component of the action argument.)

The above example illustrates one way that configuration data can be used to parameterize the signature of an automaton type; each automaton instance of this type would have a slightly different signature, according to its set of neighboring automaton instances in the configuration. A WHERE clause can be any boolean expression involving the arguments of the action, constants, and configuration data. Since the signature of an automaton is *static*, a WHERE clause cannot refer to values of state components. The set of output actions may be restricted using WHERE clauses as well.

Transition Relations Spectrum provides language constructs for defining transition relations that are similar to the "precondition-effect" notion of Lynch and Tuttle. However, there are important differences. In the precondition-effect notation, a precondition is defined for each possible action name, where an action name is taken to include the values of the action arguments. That is, Lynch and Tuttle allow the precondition to depend on the values of the arguments. This is rather impractical for a real programming language, since this might require considering each possible value of the action argument in order to determine which actions of an automaton are enabled. Considering that the data types of action arguments may have infinite domains, it would be a costly (if not impossible) procedure to evaluate the precondition for each possible action and determine the set of enabled actions.

We solve this problem by splitting the traditional precondition for an action into two parts: the *precondition* and the *selection*. In the precondition, we consider the action type as a single unit, ignoring the values of the arguments. The purpose of the precondition is to answer the following question: "Is there some assignment to the arguments of the action type that would give rise to an enabled action in the current state?" That is, if we think of each action type as a set of actions, the precondition in our language determines whether or not this set of actions contains at least one action that is enabled in the current state. Given that the precondition is satisfied, the selection clause is used to determine (possibly at random) the particular values that are assigned to the arguments of the action. Separating the argument selection from the precondition in this way avoids the impracticality of having unbound variables in the precondition. It also means that one need not select action arguments whenever the precondition is tested, but only when that particular action type is chosen for the next step of the execution.

So, the transition relation for an automaton type is defined by associating *precondition*, *selection*, and *effect* clauses with the action types listed in the signature of the automaton. Output actions may have all three kinds of clauses, while input actions have only effect clauses. To make programming easier and enhance readability, the clauses for each action type immediately follow the corresponding entry in the signature. We now say a few words about each of these three kinds of clauses. Examples of each are contained in Section 3.4.

A precondition is a predicate (or conjunction of predicates) on the current state of the automaton. Configuration data may be referenced in a precondition, but action arguments may *not* be referenced. If the precondition of an action type evaluates to *true* in a given state, then an output action of that action type is said to be *enabled* in that state.

A selection is an assignment to the argument of the action. When the argument has several components, they may be assigned separately within the selection clause. The assignments are executed sequentially and may reference (but not modify) state and/or configuration data. In addition, once an argument component has been assigned a value, it may be referenced in later assignments within the selection clause. In the current implementation, one must assign to *all* argument components, even though the WHERE clause on the output action may restrict some argument components to a single possible value; after selection, the system simply checks that the WHERE clause is satisfied by the argument selected.

An effect clause is used to derive the new state of the automaton from the old state, according to the action argument. It consists of a sequence of assignments or modifications to all or part of the state of the automaton. Again, the statements are executed sequentially and effects of earlier modifications are observed by later ones. Of course, one may reference (but not modify) the action argument in the effect clause.

Partitions So far, we have described how to define states, initial states, signatures, and transition relations of I/O automaton types. We now consider the last of the five basic components of an I/O automaton, the partition of the locally controlled actions. Since we have restricted our signatures to include only input and output actions, the locally controlled actions are simply the output actions. We divide the output actions of an automaton type into classes by placing each output action type in the signature within a CLASS block. For example, the lines

```
CLASS
   OUTPUT send
     .
     .
CLASS
   OUTPUT ack
     .
     .
   OUTPUT grant
     .
     .
```

say that for each instance of this automaton type, all `send` actions are in one class of the partition and all `ack` and `grant` actions are in another class. The class block construct is simple, and makes the division of actions into classes obvious at a glance. However, as we have described it so far, the construct is not quite general enough, since all output actions of a given type must belong to the same class. Sometimes one wishes to place different actions of the same type into different classes, according to their argument values. Therefore, we allow a class block to be *parameterized* using configuration data. Each parameter may take on values from a fixed set, and the type of the parameter is inferred to be the type of the elements of that set. Just as we use `s` to refer to state components and `a` to refer to action arguments, we use `c` to refer to class parameters. For example,

```
CLASS (dest:neighbors(self()))
   OUTPUT send WHERE eq(a.to, c.dest)
```

declares several classes, one for each neighbor of the automaton instance in the configuration graph. Since `neighbors(self())` is a set of automaton id's, `dest` has type automaton_id. Each class contains a set of `send` actions with that neighbor as the "to" component of the argument. A class may contain more than one type of action, and may be parameterized by more than one set. Although the number of classes must be finite, the number of actions within a class may be infinite.

In defining a class, one may optionally specify a non-negative integer as the "weight" of that class. In the current implementation, these weights are interpreted by the randomized scheduler as the average relative speeds of the processes. For example, if two classes continually contain enabled actions, and the weight of the first class is twice that of the second, then the first class will take twice as many steps as the second, on average.

3.4 Example

As a simple example of the use of the language, we present an implementation of LeLann's algorithm for electing a leader in an asynchronous ring, where each process in the ring starts with a unique identifier [19]. Essentially, each process passes a message containing its identifier to its left neighbor in the ring. Processes forward only those messages containing identifiers greater than their own. The process whose identifier travels all the way around the ring announces that it is the leader. To model the asynchrony of message delivery, we place a channel automaton between each pair of neighbors in the ring. The automaton types *channel* and *process* are shown in Figure 1. The user-supplied names in the configuration are used as the process identifiers. (In a configuration, the default user-name of a process is its system-supplied automaton-id converted to a string.)

Each automaton has an input action called *initially*. The action is not an output of any automaton, but the system causes an initially event to occur once at the beginning of each execution to initialize the state of each automaton. Uninitialized state components are assigned default values, as described earlier.

In this example, each type of output action is in its own class. However, we could just as easily place the *send* and *leader* actions in one class. Alternatively, one might *parameterize* the class containing the *send* action as shown in Figure 2. In that figure, x is declared to be of type automaton_id, since the all_of_type function returns a set of automaton id's. For each element of all_of_type("process"), a separate class is created with x bound to that element. Thus, each possible *send* action would be in its own class of the partition.

3.5 Support for Verification, Analysis, and Visualization

So far, we presented Spectrum language constructs that allow one to express algorithms as I/O automaton types. But since the language is part of an algorithm development tool, we would like more than just the ability to express algorithms. We would like the language to provide support

```
DATA   message tuple(msg:string, chan:automaton_id)
DATA   buffer  multiset(message)

ACTION initially ()
ACTION send    message
ACTION receive message
ACTION leader  string

AUTOMATON channel
   STATE buffer
   INPUT initially
      EFF mset_init(s)
   INPUT send WHERE eq(a.chan,self())
      EFF mset_insert(s,a)
   CLASS
      OUTPUT receive
         PRE bool_not(mset_empty(s))
         SEL assign(a,mset_random(s))
         EFF mset_delete(s,a)

AUTOMATON process
   STATE tuple(pending:set(string), status:string)
   INPUT initially
      EFF assign(s.pending,set_single(name(self())))
          assign(s.status,"waiting")
   INPUT receive WHERE set_el(in(self()),a.chan)
      EFF ifthenelse(greater(a.msg,name(self())),
                     set_insert(s.pending,a.msg),
                     ifthen(eq(a.msg,name(self())),
                            assign(s.status,"elected")))
   CLASS
      OUTPUT send
         PRE bool_not(set_empty(s.pending))
         SEL assign(a.msg,set_random(s.pending))
             assign(a.chan,set_random(out(self())))
         EFF set_delete(s.pending,a.msg)
   CLASS
      OUTPUT leader
         PRE  eq(s.status,"elected")
         SEL  assign(a,name(self()))
         EFF  assign(s.status,"announced")
```

Figure 1: Automaton types for LeLann's leader election algorithm.

```
CLASS (x: all_of_type("process"))
   OUTPUT send
      PRE set_el(s.pending,name(x))
      SEL assign(a.msg,name(x))
          assign(a.chan,set_random(out(self())))
      EFF set_delete(s.pending,a.msg)
```

Figure 2: A parameterized class.

for studying algorithm executions. In this section, we present language constructs that are useful for algorithm verification, analysis, and visualization.

3.5.1 State Invariants

Constructing an *assertional* proof is a common method for showing that an algorithm meets its specification. In an assertional proof, one states a number of properties on the state of the system that imply the correctness of the algorithm. Then, one shows, usually by induction on the length of the execution, that these properties are *invariant*. That is, they hold in all reachable states of the algorithm. Often, the most difficult part of these proofs is in coming up with the right set of invariants. Trying to construct a proof using the wrong invariants can result in much wasted effort. It is helpful to be able to check invariants automatically on algorithm executions in order to have an opportunity to refine them before proceeding with a rigorous proof. Therefore, the Spectrum programming language provides the ability to specify a set of invariants on the state of an automaton that are to be checked after each step of the execution. For example, the clause

```
INVARIANT
   less(s.status,5)
   bool_or(greater(s.status,0), set_empty(s.buff))
```

specifies that the *status* component of the state must always be less than five, and that either *status* is greater than zero or *buff* is empty. After each step of the automaton's execution, the set of invariants is checked, and the execution is interrupted if an invariant is violated.

3.5.2 Spectators

Assertional proofs use invariants on the state of a computation primarily as a means to show properties of the *behavior* of an algorithm. That is, our main concern is not with the states themselves but with determining that the sequence of actions that occurs at the boundary between the algorithm and the environment is consistent with the problem specification. As we saw in Section 2, the I/O automaton model provides *schedule modules* as a way to specify problems in terms of a set of allowable behaviors. In this section, we describe a device called a *spectator* that allows one to check executions of algorithms against the set of allowable behaviors specified by a schedule module.

For purposes of illustration, we define a schedule module for the mutual exclusion problem. Fix n, a positive integer, and let $\mathcal{I} = \{1, 2, \ldots, n\}$. We define schedule module M with *sig*(M) as follows:

Inputs:	$\text{UserTry}_i, i \in \mathcal{I}$	Outputs:	$\text{Crit}_i, i \in \mathcal{I}$
	$\text{UserExit}_i, i \in \mathcal{I}$		$\text{Rem}_i, i \in \mathcal{I}$

Schedule module M interacts with an environment that may be thought of as a collection of n user processes u_i, $i \in \mathcal{I}$, where each process u_i has outputs UserTry_i and UserExit_i, and has inputs Crit_i and Rem_i. A UserTry_i action means that process u_i wishes to enter its critical section. A Crit_i action by M gives u_i permission to enter its critical section. A UserExit_i action means that process u_i is leaving its critical section. Finally, the Rem_i action gives u_i permission to continue with the remainder of its program. If β is a sequence of actions of M, then we define $\beta|i$ to be the subsequence of β containing exactly the UserTry_i, Crit_i, UserExit_i, and Rem_i actions. Before defining the allowable schedules of M, we define the set of well-formed sequences of actions of M. Let β be a sequence of actions in *sig*(M). We say that β is *well-formed* iff for all $i \in \mathcal{I}$, all prefixes of $\beta|i$ are prefixes of the infinite sequence UserTry_i, Crit_i, UserExit_i, Rem_i, UserTry_i, $\text{Crit}_i, \ldots$. This says, for example, that a process will not issue a try request while in its critical section.

We define the set *scheds*(M), the allowable external behaviors of M, as follows. Let β be a sequence of actions in *sig*(M). Then $\beta \in \textit{scheds}(M)$ iff the following conditions hold:

1. M preserves well-formedness in β.
2. If β is well-formed, then $\forall i, j \in \mathcal{I}$, if Crit_i and Crit_j occur in β (in that order), then UserExit_i occurs between them.

Condition (2) says that no two processes are in their critical sections simultaneously, provided that the user processes preserve well-formedness.

We would like to write a spectator to check executions of an automaton system against the allowable behaviors specified by schedule module M. A spectator is simply an I/O automaton with no output actions that observes the actions taken by other automata. By writing spectators without output actions, we need not be concerned that a spectator could interfere with the execution of an algorithm. Furthermore, it is not sensible to have spectators report a detected error by means of an output action, because the scheduler might not give the spectator a chance to take a step until much later in the execution. Instead, we write a spectator so that one of its own invariants is violated whenever it detects an error. Conveniently, this interrupts the simulation immediately, so that the user may explore the source of the error. One can usually construct a spectator directly from the schedule module specifying the problem.

The spectator in Figure 3 corresponds to Condition 2 of schedule module M, and could be used to check the executions of a mutual exclusion algorithm. In this spectator, the state component `last-crit` keeps track of the index of the process most recently in the critical section, and the component `in-crit` keeps track of whether the last input action was Crit or UserExit. When a Crit action occurs, the invariant $ok = \textit{true}$ is violated if and only if no UserExit occurred since the

```
AUTOMATON CheckMutex
   STATE tuple(last-crit: integer, in-crit: boolean, ok: boolean)
   INVARIANT eq(s.ok,true)
   INPUT initially
      EFF assign(s.last-crit, 0)
          assign(s.in-crit, false)
          assign(s.ok, true)
   INPUT Crit
      EFF assign(s.ok, bool_not(s.in-crit))
          assign(s.last-crit, a)
          assign(s.in-crit, true)
   INPUT UserExit
      EFF assign(s.ok, bool_and(s.in-crit, eq(s.last-crit,a)))
          assign(s.in-crit, false)
```

Figure 3: A spectator for mutual exclusion.

last preceding Crit action. When a UserExit action occurs, the invariant is violated if and only if the argument of the action is not the index of the process currently in the critical section. It is easy to see how these cases are derived from Condition 2 of schedule module M. One could write a similar spectator automaton type for checking that each user's execution is well-formed.

Note that a spectator depends only on the problem specification, and never on the algorithm itself. That is, a spectator for a given problem specification could be used to check *any* solution to that problem. In addition to verifying that executions are correct, spectators can be helpful in the analysis of algorithm efficiency. For example, one might use a spectator to count the number of messages sent in an execution, or to keep track of the rates at which processes enter their critical sections in a mutual exclusion algorithm. Again, because a spectator has no output actions, we know that such analysis cannot interfere with the algorithm execution.

3.5.3 Pseudovariables

Another language construct provided in Spectrum is the MAINTAIN clause, which updates state components after every action of an automaton. The MAINTAIN clause is somewhat similar to Lamport's *state functions* [18] and the ALWAYS construct of UNITY [4]. It is used to maintain "pseudovariables," variables that are a function of the remaining state components. For example,

```
MAINTAIN
   assign(s.status, set_size(s.buff))
```

would cause the state component *status* to be updated to the buffer size after every step of the automaton.[4] The above is a relatively simple example, but a pseudovariable can be used to sum-

[4]The MAINTAIN clause is executed after the effects clause of each action and before any local invariants are checked. If a MAINTAIN clause contains multiple statements, they are executed sequentially.

marize the state of an automaton in arbitrarily complicated ways. When pseudovariables take on integer or boolean values, they can then be used as the basis of algorithm visualization. As we will see in Section 5, each automaton instance is represented in the graphical user interface as an icon. Within the interface, one can create "summary mappings" that associate state components of an automaton type with the colors of the icons. In this way, important automaton state information can be displayed during simulation. Using the above example, one could create a summary mapping for the state component *status* and watch the sizes of the buffers of each automaton grow and shrink as the execution proceeds; such a visualization might be useful for identifying congestion in parts of the network being simulated.

MAINTAIN clauses are also useful for algorithm analysis. For example, one might place a statement in the MAINTAIN clause to increment a counter whenever the automaton takes a step. By keeping track of such information in the MAINTAIN clause, rather than dispersing it throughout the transition relation, one can separate those parts of the code that concern the algorithm itself from those that are present only for the purposes of visualization or analysis.

4 The Simulator

In this section, we present the Spectrum simulator, the second main component of the Spectrum Simulation System. Given a collection of automaton types and a configuration, the simulator performs all of the functions necessary to load and type-check the automaton types, initialize the simulation, interpret automaton state information, evaluate expressions in the transition relation, and monitor the set of enabled classes in order to produce executions of an I/O automaton system. The simulator provides a choice of scheduling options, as well as services for checking state invariants, updating state information on the display, rolling back executions, and generating trace files.

The input to the simulator consists of a collection of automaton types and a configuration. The automaton types are presented to the simulator in the form of a text file containing code written in the Spectrum programming language. The configuration is made available to the simulator as a data structure that is shared with the user interface. In addition to these two pieces of input, the simulator may receive input from the user during the course of the simulation session. We concentrate here on the functionality of the simulator. Implementation details are discussed elsewhere [8].

The simulator has four main logical components, the loader, the interpreter, the execution loop, and the scheduler. These components work together to implement the semantics of the Spectrum programming language, thereby capturing the semantics of the I/O automaton model. We now discuss each of the components in turn.

The *loader* parses and type-checks the automaton types file, and organizes the automaton type definitions into data structures that are used by the interpreter. Error messages are generated for any type errors or syntax errors encountered, as well as for violations of restrictions on various clause types. For example, one such violation is a precondition containing a statement that could

potentially change a state component. When a function expression is parsed, the loader creates a data type structure representing the type of the return value of the function. For polymorphic functions, the return type is determined from the types of the arguments. This structure is carried with the expression for further type checking (e.g., when the expression is an argument to another function) and for use by the interpreter.

The job of the *interpreter* is to evaluate expressions represented as automaton types data structures in order to provide four basic services: determine whether or not any action is enabled from a given class in a given state, select an action from an enabled class[5] in a given state, determine which automata have a given action as an input, and produce a new state of an automaton, given its old state and an action in its signature.

In order to determine whether an action of a given type is enabled in the current state of an automaton, the interpreter evaluates the precondition for that action. Similarly, to assign to the arguments of an action, the interpreter evaluates the selection clause for that action. In order to determine the set of participant automata for an action, the interpreter checks the WHERE clause of that action for each automaton instance having that action type in its input signature. Finally, to make state transitions, the interpreter evaluates the EFF clause of the action for each participant. All such calls to the interpreter are under the control of the execution loop, presented next.

In the course of evaluating expressions, no type errors can occur, since the loader performs static type checking. However, some errors are still possible, such as division by zero or drawing a random element from an empty set. When such a situation arises, the simulator generates an error message that indicates the nature of the error, the name of the action being processed, and the id of the offending automaton.

The *execution loop* manages the entire simulation. Upon invocation, the execution loop assumes that the automaton types data structures and configuration data structures are already in place. Its first task in starting up a simulation is to initialize the data structures for each of the automaton instances and their classes. The execution loop then calls on the interpreter to execute the "initially" action at all automaton instances whose automaton types have that action in their input signatures. This serves to initialize the states of the automata. Following this, each class in the system is checked to determine whether or not it contains an enabled action. That is, for each class of each automaton instance, the interpreter is called to evaluate the precondition for each action in the corresponding class of the automaton type definition. If any action in a class is found to be enabled, then the class is marked as being enabled and the scheduler, to be discussed next, is informed of each class so enabled. This completes the initialization procedure.

A high-level description of the execution loop is shown in Figure 4. At each iteration through the loop, the scheduler is asked to produce the next class to take a step. (If no classes contain enabled actions, the simulation terminates.) Then, an enabled action is chosen from that class. Since we are not required to be fair to the actions *within* classes, but only to the classes themselves, we arbitrarily choose the first enabled action in the class. Then, the selection clause (SEL) for that action is

[5] A class is *enabled* in a given state if some action in the class is enabled from that state.

```
create the automaton instance data structures and classes
execute the "initially" action at each automaton
determine the set of enabled classes and inform scheduler
while the set of classes containing enabled actions is nonempty
    ask scheduler for the next class to perform an output
    choose an enabled action from that class
    execute the SEL clause to determine the arguments
    determine the set of participant automata
    for each participant,
        produce a new state of that automaton based on the event
        check invariants
        determine the new set of enabled classes of that automaton
        inform the scheduler of updates to the set of enabled classes
    update the display, write to the trace file, etc.
```

Figure 4: Pseudocode for the execution loop.

evaluated to determine the action arguments. Following this, the set of participants in the action is determined: for each automaton having that action type in its input signature, the corresponding WHERE clause is evaluated; if it evaluates to true, then that automaton is a participant in the action. For each participant, the EFF and MAINTAIN clauses are evaluated to produce the new state of the automaton. At this point, any invariants on the state are checked and the execution is interrupted if violations are discovered. Since the state of each participant may change, the set of enabled classes of each participant is recomputed, and the scheduler is informed of any changes. At the end of each iteration, the user interface, described in the next section, is informed of both the action and the set of participants in order to update the display.[6] Additional features provided by the simulator include the generation of formatted trace files containing schedules and selected state information, and the ability to back up or advance the simulated execution to an arbitrary step number.

The fourth component of the simulator is the *scheduler*. The scheduler maintains a list of the classes containing enabled actions, and is called on by the execution loop at each step in the execution to choose the next class to be given a turn. This choice is made according to a particular scheduling algorithm, selected by the user before simulation is begun. In the current implementation of the simulator, there is a choice of two scheduling algorithms, randomized and round robin.

The randomized scheduler makes use of the weights on each of the automaton classes. It keeps track of the total t of the weights of all enabled classes, and at each step in the execution selects a class with weight w with probability $\frac{w}{t}$. The round robin scheduler treats the list of classes with

[6] In fact, it is the user interface that requests each iteration of the loop, since the user may wish to interrupt the simulation in order to study the states of automata, change the visualization, roll back the execution, etc.

enabled actions as a queue. It always selects the first class in the list and moves that class to the end of the list. Newly enabled classes are always added to the end of the list. Recall that the I/O automaton model's fairness definition requires that if a class continually contains an enabled action, then eventually an action occurs from that class. According to this definition, the round robin scheduler guarantees fairness. The randomized scheduler, on the other hand, only produces fair schedules with high probability.

The way that scheduling is accomplished in Spectrum constitutes a major difference between the present work and related work in the area of discrete event simulation. In discrete event simulation (see Misra [28]), an "events list" is kept for all of the actions that should be simulated, but have not yet occurred in the simulation. Each event in the system may cause new events to be added to the events list. An important difference between the events list and the class list maintained by our scheduler is that no event is removed from the events list until it occurs, while events in an I/O automaton system may cause actions to become disabled and thus cause classes to be removed from the scheduler's class list. In general, the semantics of the I/O automaton model are such that any action may become disabled by the occurrence of other events.

5 The User Interface

The Spectrum user interface is used to build configurations of automaton systems, and to control simulation and visualization of those systems. To simulate an algorithm in Spectrum, one first specifies the various automaton types of the system using the Spectrum language. Then, one graphically constructs a system configuration (of the sort described in Section 3.2). In *configure mode*, the user interface provides tools for building and editing a system configuration. Recall that a configuration specifies, for each automaton in the system, the type of that automaton, a unique system-supplied identifier and a user-supplied name, and a set of adjacent edges in a directed graph. Icons of different shapes are used to represent the different automaton types. In configuring I/O automaton systems, one uses the mouse to create instances of automaton types, connect them with directed edges, and arrange them spatially. The edges are useful for breaking symmetry, defining communication patterns, or establishing other relationships between automata. The sets of incoming and outgoing edges of automaton instances are accessible in automaton type definitions, as described in Section 3. Other editing options include assigning names to automaton instances (apart from the system-supplied automaton id), deleting instances and edges, and changing the type of an instance.

One may also specify "summary mappings" in configure mode. These mappings associate the colors of automaton instance icons with the values of the state components of those instances. Each instance icon is divided into two parts, a border and a center. One may map a single state component to both the border and center, or may map a different state component to each. MAINTAIN clauses (see Section 3) are useful for updating state components that are used in summary mappings. Summary mappings are established for an entire automaton type, rather than on an instance by instance basis. The resulting uniformity makes the visualization easier to

understand. Summary mappings are specified in configure mode in preparation for visualization of algorithm executions. However, one is free to change summary mappings during simulation, as well.

An important configuration option is the ability to build new automaton types by composing others. For example, one might model an asynchronous system by associating a message buffer automaton with each user process automaton. Rather than explicitly creating each pair of automaton instances, one might define a new automaton type that is the composition of a user process automaton and a message buffer automaton. One may create instances of an automaton type before that type is (fully) defined. Whenever a composed type is defined or modified, the changes are immediately reflected in all instances of that type. Hierarchical composition is supported; components of a composition may be compositions themselves. However, it is illegal to create recursive type definitions. For example, if type A is the composition of types B and C, then neither B nor C may have A as a component.

Having specified a set of automaton types and a configuration, one can run simulations of the I/O automaton system in *simulate mode* of the user interface. As the simulation runs, the display is updated as follows. After each step of the execution, the borders around the set of automata involved in that step are highlighted (red for output and blue for input) and the action name with its argument value is displayed. In addition, the colors of the automaton instance icons are updated according to the summary mappings. In order to provide flexible exploration of algorithms, the summary mappings may be changed during simulation.

Taking advantage of the structure provided by I/O automaton composition, the Spectrum interface allows one to look inside an automaton to see its components. This allows one to view algorithm simulations at different levels of detail. If the automaton being "opened" is an instance of a composed type, then a the user interface creates a window containing the automaton's components, colored according to their summary mappings, However, if the automaton is an instance of a type explicitly defined in the automaton types file, then a window created that contains a textual display of the values of the automaton's state variables. All state information (whether graphical or textual) is updated as the simulation runs.

To facilitate close study of algorithms, the user interface allows one to invoke special simulator functions. For example, one may roll back or advance the execution to any arbitrary step number, and may generate a trace file containing the schedule (sequence of actions) of the execution and selected state information. Errors, such as violations of local state invariants, interrupt the simulation so that the user may explore possible causes.

The interface is written in standard C [17] on top of the X11 window system [32] and requires four color planes. The current implementation runs on DEC Microvaxes. A three-button mouse is used for most interface commands.

6 Conclusion

We conclude this paper with an evaluation of Spectrum in terms of the design goals outlined in Section 1, taking into account the experiences of Spectrum users. For each design goal, we review those aspects of the system that were designed to help achieve the goal, and consider how successful these actually were. In the course of this evaluation, we suggest a number of possible directions for further work.

The design must be faithful to a formal model. The Spectrum programming language provides mechanisms for defining the signatures, states, initial states, transition relations, and partitions of I/O automata. The user interface provides facilities for building up complex systems of I/O automata using standard I/O automaton composition, preserving the essential process structure of the components in the composition. And the Spectrum interpreter generates executions of these systems that are consistent with the semantics of the I/O automaton model. However, Spectrum provides what should properly be called a *subset* of the features of the I/O automaton model. Due to physical limitations of the digital computer, we were forced to abandon the nondeterminism present in the formal model and replace it by randomization at several places in the system design. Also due to practical considerations, the infinite collections of automata allowed in the formal model are not supported in Spectrum.

The nondeterminism present in the I/O automaton model is not of the usual complexity-theoretic flavor. Instead, nondeterminism is used in order to make algorithms and their correctness proofs more *general*. Rather than present a particular deterministic implementation of an algorithm, one describes a (nondeterministic) I/O automaton that, in some sense, embodies all such deterministic implementations. By carefully introducing randomized functions in the language (such as selecting a random number or choosing a random element from a set) and providing a randomized scheduler, we are able to support algorithm descriptions that are quite general. Furthermore, we have found that the random executions generated by Spectrum are quite useful for gaining insight into how an algorithm works, finding errors in an algorithm, and learning about an algorithm's performance. Even so, one cannot possibly hope to automatically generate all of the strange executions that could cause an algorithm to fail. Simulating algorithms in the Spectrum system can provide help in understanding and debugging the algorithm, but unless one can generate all possible executions of an algorithm, it will always be necessary to construct a formal correctness proof. This is why it is so valuable that the semantics of the Spectrum language is consistent with that of the formal I/O automaton model.

In Spectrum, the configuration is static. That is, all automaton instances are in existence at the beginning of the execution. This limitation is a problem when one wishes to consider algorithms that involve dynamic process creation. In the I/O automaton model, a system consists of a static collection of components, but the number of components may be infinite. Therefore, one can model dynamic process creation by assuming that all possible processes exist at the beginning of a computation and then "waking them up" as the algorithm proceeds. (For examples of this, see [21], [23] and [20].) Since supporting an infinite collection of automata is a physical impossibility, an

interesting direction might be to extend Spectrum with mechanisms to support dynamic automaton creation.

The language should be natural for expressing a large class of distributed algorithms. Clearly, the choice of the formal model on which a language is based makes a great impact on the expressive power of that language. Because of its separation of inputs and outputs, its treatment of fairness, and its compositionality properties, the I/O automaton model is particularly well suited for describing a wide class of distributed algorithms, and has been the basis for much research in that area. By making the I/O automaton model the theoretical basis of the Spectrum system, we were able to take advantage of the expressive power provided by that model. Virtually any message-passing algorithm may be expressed as an I/O automaton. Spectrum's rich set of data types and built in operators make it easy to express a wide variety of distributed algorithms. And since one can model the communication system explicitly as a separate I/O automaton, it is easy to express (and simulate) algorithms that have widely varying assumptions about the underlying network.

However, the expressive power of the Spectrum language could be improved in some areas. Although the I/O automaton model provides excellent support for modelling message-passing algorithms, many of the important asynchronous concurrent algorithms are described using shared memory. And in some cases one might wish to use both shared memory and message passing to describe different parts of an algorithm. In [6], we present extensions to the I/O automaton model for describing shared memory algorithms. We are planning extensions to Spectrum based on this work.

Another property of the I/O automaton model is that system components cannot observe the private states of other components. But sometimes one wishes to describe distributed algorithms as layers of modules such that the higher layers are allowed to observe (but not modify) the variables of the lower layers. One layering mechanism, called *superposition*, is defined by Chandy and Misra for the UNITY programming language [4]. It is essentially a program transformation that adds a layer on top of a program by introducing a set of higher level variables and code that makes use of them. The transformation is required to preserve all the properties of the underlying program. In [9], we extend the I/O automaton model to permit superposition of program modules. We are working on ways to incorporate these extensions into Spectrum.

The expressive power of Spectrum for describing distributed algorithms based on message passing is quite good. However, some Spectrum users (particularly those without prior experience with I/O automata) find that it is difficult to "think" in the Spectrum language [11]. Apparently, this problem is due to the syntax used for expressions and the lack of infix operators. A sugared version of the syntax and the addition of user-defined data types would probably solve this problem.

The system should encourage experimentation with algorithms. The decision to separate automaton types from the system configuration allows Spectrum users to experiment with a given algorithm in a variety of system configurations. Furthermore, the fact that systems are configured graphically and their executions are observed visually saves time and effort in setting up and analyzing simulations. Crucial to making the separation work is the ability to parameterize

an automaton type according to the configuration data. For example, one can use the edges of the configuration graph to represent communication paths in a network and parameterize an automaton type on its sets of incoming and outgoing edges. This automaton type could be instantiated many times in a configuration, each time with a different number of incoming and outgoing channels.

The summary mappings of the user interface and the MAINTAIN clause of the Spectrum language work together to provide a simple, yet flexible, mechanism for handling algorithm visualization. There are two general methodologies for program visualization: declarative and imperative [31]. In imperative visualization, one embeds procedure calls in the algorithm being studied in order to effect changes in the display, while the declarative approach establishes relationships between state information of the algorithm and objects on the display. Summary mappings are declarative. This encourages experimentation, since it allows visualizations to be set up and modified quickly without changes to the algorithm itself.

The modularity provided by I/O automaton composition is also useful for experimentation. One can easily substitute one module for another in the configuration without having to return to the automaton types definitions. For example, one might write two versions of a automaton, one that is faulty and one that is not, and then experiment with the fault tolerance of an algorithm simply by changing the types of automaton instances in the configuration.

An important part of experimenting with algorithms is generating many different executions for them. In Spectrum, there are currently two scheduling algorithms, randomized and round robin, as described in Section 4. Permitting user intervention in the scheduling process, however, would improve the flexibility of the system. One might allow the user to select the next class to take a step, or one might add mechanisms for expressing algorithm-dependent scheduling rules, such as "if any automaton has fifty messages in its buffer, then give it priority to take a step." Another means of intervening in simulation would be to permit users to change the values of automaton state components during simulation.

Section 3 highlighted two mechanisms for studying algorithms from a correctness proof point of view: the INVARIANT clause and the spectator. These are useful for experimenting with an algorithm by generating many (or long) executions and mechanically checking for violations of the invariants or the problem specification, and for computing performance statistics such as the number of messages generated in an execution. Since the verification and analysis mechanisms are kept separate from the algorithm being studied, one need not modify (or clutter up) the algorithm in order to check or analyze its executions. The execution rollback feature and the trace file capability are particularly helpful for analyzing executions that result in violations of the invariants or the specification. But the INVARIANT mechanism is not quite strong enough for the study of distributed algorithms. Typically, when one constructs proofs of distributed algorithms, one thinks not only about *local* invariants on the private states of individual components, but also about *global* invariants that involve the states of many system components. In the current Spectrum implementation, global invariants cannot be checked. However, extending Spectrum with a superposition operator will not only enhance the expressive power of the Spectrum language, but will also permit one to write (and check) global invariants simply by writing invariants of the higher layer automa-

ton that involve the variables of the components of the lower layer. In addition, superposition will allow one to create summary mappings based on aggregate information about components of a composition.

Using spectators, one can check the correctness of an execution and study the message complexity of an algorithm, but it is difficult to study time complexity in Spectrum. However, the I/O automaton model has recently been extended for the study of real-time systems, and has been used to construct timing-based proofs for distributed algorithms [1, 2, 22, 29]. This work could form the basis of useful extensions to Spectrum. Timing constraints would be associated with automaton *classes* and handled automatically by the system.

Another way to improve the utility of a simulation system for running experiments is simply to make the simulator run faster. One way to accomplish this is to introduce concurrency in the simulator itself. We have developed an algorithm for achieving highly concurrent distributed simulation of I/O automata [7], and plan to incorporate this capability in a future version of Spectrum.

The design should achieve economy and integration. Although the Spectrum design separates logically independent concerns, the system is well-integrated. We have achieved our goal of using the same language mechanisms for writing programs, creating debugging tools, specifying invariants, and setting up visualization. In addition, a single graphical interface is used for both constructing the system configuration and controlling the simulation.

We stated in the introduction that an aim of this work is to demonstrate how distributed algorithm specification, design, debugging, analysis, and proof of correctness may be integrated within a single formal framework. We have described the Spectrum Simulation System based on the I/O automaton model. By remaining faithful to that model, we have eased the transition between formal specifications and algorithm descriptions, and also the transition between algorithm descriptions and correctness proofs. Similarly, we have provided specific tools for using problem specifications for checking algorithm executions mechanically (spectators), and a language construct for expressing invariants on program states. These tools can be used to help integrate the design and debugging process with the construction of a formal correctness proof, and they provide a foundation for further integrating the tasks of developing algorithms and proving their correctness.

Acknowledgments

I thank Nancy Lynch for many useful discussions about the Spectrum design, Bill Weihl for his suggestions on the presentation of the ideas, and Christopher Colby for implementing the Spectrum user interface. I also thank Hagit Attiya, Alan Fekete, Greg Troxel, and Mark Tuttle for their comments on the system as it was being developed. And I sincerely appreciate the comments and suggestions from the first users of Spectrum: Mini Gupta, John Leo, Stephen Ponzio, and Isaac Saias, and the students in Nancy Lynch's graduate course in distributed algorithms (Fall Semester, 1990).

References

[1] Hagit Attiya and Nancy Lynch. Time bounds for real-time process control in the presence of timing uncertainty. In *Proceedings of the 10th Real Time Systems Symposium*, December 1989.

[2] Hagit Attiya and Nancy Lynch. Using mappings to prove timing properties. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, August 1990.

[3] Bard Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computing, Special Issue on Parallel and Distributed Algorithms*, 37(12):1506–1514, December 1988. Also in 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August, 1987, pp. 249-259.

[4] K. Mani Chandy and Jayadev Misra. *A Foundation of Parallel Program Design.* Addison–Wesley, Reading, MA, 1988.

[5] Alan Fekete, Nancy Lynch, and Liuba Shrira. A modular proof of correctness for a network synchronizer. In *The 2nd International Workshop on Distributed Algorithms*, July 1987. Amsterdam, The Netherlands.

[6] Kenneth Goldman and Nancy Lynch. Modelling shared state in a shared action model. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[7] Kenneth J. Goldman. Highly concurrent logically synchronous multicast. In *Proceedings of the 3rd International Workshop on Distributed Algorithms, Nice, France, Lecture Notes in Computer Science 392.* Springer–Verlag, September 1989. Also, MIT Technical Memorandum MIT/LCS/TM–401.

[8] Kenneth J. Goldman. Distributed algorithm simulation using Input/Output Automata. Technical Report MIT/LCS/TR–490, MIT Laboratory for Computer Science, July 1990. Ph.D. Thesis.

[9] Kenneth J. Goldman. A compositional model for layered distributed systems. Submitted for publication, February 1991.

[10] Kenneth J. Goldman and Nancy A. Lynch. Quorum consensus in nested transaction systems. In *Proceedings of the 6th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 27–41, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR–390.

[11] Aparna M. Gupta. I/O automaton based simulation of selected distributed algorithms. Senior Thesis, MIT Laboratory for Computer Science, June 1990.

[12] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, April 1990.

[13] David Harel. On visual formalisms. Technical Report CMU–CS–87–126, Carnegie Mellon Computer Science Department, June 1987.

[14] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[15] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[16] INMOS. Transputer reference manual and product data. INMOS Limited, September 1985.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[18] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[19] G. LeLann. Distributed systems — towards a formal approach. In *Information Processing 77 (IFIP)*, pages 155–160, Toronto, 1977. North Holland Publishing Co., Amsterdam.

[20] John Leo. Dynamic process creation in a static model. M.S. Thesis, MIT Laboratory for Computer Science, May 1990.

[21] N. Lynch and M. Merritt. Introduction to the theory of nested transactions. In *International Conference on Database Theory*, pages 278–305, Rome, Italy, September 1986. Also, expanded version in Technical Report, MIT/LCS/TR-367, MIT Laboratory for Computer Science, July 1986. Revised version in *Theoretical Computer Science*, 62(1988):123-185.

[22] Nancy Lynch. Modelling real-time systems. In *Foundations of Real-Time Computing Research Initiative*, pages 1–16, November 1988. ONR Kickoff Workshop.

[23] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* In progress.

[24] Nancy A. Lynch and Kenneth J. Goldman. Distributed algorithms. Technical Report MIT/LCS/RSS-5, MIT Laboratory for Computer Science, May 1989. MIT Research Seminar Series.

[25] Nancy A. Lynch, Yishay Mansour, and Alan Fekete. Data link layer: Two impossibility results. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 149–170, August 1988.