Distributed Algorithm Simulation Using Input/Output Automata

by

Kenneth J. Goldman

S.M. EECS, Massachusetts Institute of Technology (1987)
Sc.B. Computer Science, Brown University (1984)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1990

Signature of Author.. .....
Department of Electrical Engineering and Computer Science
July 16, 1990

Certified by .....
Nancy A. Lynch
Professor
Thesis Supervisor

Accepted by . .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Distributed Algorithm Simulation Using Input/Output Automata

by

Kenneth J. Goldman

Submitted to the Department of
Electrical Engineering and Computer Science on July 16, 1990
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

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.

Motivated by a desire to broaden the class of algorithms that may be studied using Spectrum, we propose two extensions to the I/O automaton model. First, we extend the I/O automaton model to allow modelling of shared memory systems, as well as systems that include both shared memory and message passing communication. This extension supports description, verification, and analysis of shared memory systems. As an example, Dijkstra's classical shared memory mutual exclusion algorithm is presented and proved correct. In addition, we illustrate how the extended model provides a unified formal framework in which shared memory systems and message passing systems may be related. Second, we extend the I/O automaton model with a *superposition* operator that permits system modules to be combined in layers so that higher layers may observe (but not modify) the variables of lower layers. We show that superposition does not affect the set of executions of the underlying module, thus preserving all properties of that module. A formal specification mechanism is presented that allows the set of correct behaviors of the higher level module to be expressed in terms of the states of the underlying module. As an illustration of the superposition extension, the global snapshot algorithm of Chandy and Lamport is presented with a complete proof of correctness. For both of these model extensions, we propose corresponding extensions to the simulation system.

The final contribution of this thesis is a distributed algorithm that may be used to achieve distributed simulation of algorithms written as I/O automata. The algorithm solves a new synchronization problem, *logically synchronous multicast*, that captures the synchronization semantics of the I/O automaton model.

Keywords: Distributed systems, distributed algorithms, formal models, I/O automata, programming languages, simulation, program visualization, algorithm development, correctness proofs, shared memory, mutual exclusion, linearizability, superposition, global snapshot, multicast, synchronization, distributed simulation.

Thesis supervisor: Nancy A. Lynch

Title: Professor

Acknowledgments

I could not have asked for a better research advisor than Nancy Lynch. She has been a constant source of research ideas, and has provided direction and encouragement at all the right moments. Nancy is a co-author of Chapter 7, and has been involved with the details of many of the other chapters. I also thank the other members of my thesis committee, Baruch Awerbuch and Bill Weihl, whose comments have contributed to the quality of this thesis.

The work in Chapter 7 on proofs for shared object systems is joint work with Kathy Yelick. I also thank Kathy for her careful reading of the remainder of that chapter.

Many of the past and present members of the Theory of Distributed Systems research group have contributed to this thesis, and all have made this group an exciting place to do research. I thank Christopher Colby for implementing the Spectrum user interface, and for his patience when I would occasionally decide to change the specification. I thank John Leo, Stephen Ponzio, and Mini Gupta for their comments on using the Spectrum Simulation System. In addition, I thank thank Alan Fekete for several discussions during the early part of this work, Jennifer Welch and Mark Tuttle for their detailed comments on both the technical details and the presentation of the logically synchronous multicast algorithm, and Hagit Attiya for several technical discussions. Also, I thank Anna Wiseman for taking care of all that paperwork.

I am grateful to Tom Miller and Paris Kanellakis, who introduced me to computer science research and were influential in my decision to attend graduate school.

I thank my parents, Lester and Judy, for all the love and support they have given me during the past 27 years. I thank the rest of my family for their moral support, and for not asking too many times when I would graduate. In particular, I thank my aunt and uncle, Phyllis and Alfred Schneider, for being a second family to me during my stay in the Boston area. I thank my in-laws, Robert and Marilyn Goldwasser, for their frequent phone calls and visits, for their words of encouragement, and especially for their daughter.

This thesis would not have been possible without the help of my wife, Sally. I thank her for emotional and technical support, for taking on extra responsibilities whenever I faced a deadline, and for never doubting once that we would finish at the same time. Finally, I thank our son, Mark, for providing energy, excitement, and humor whenever it was needed most.

This research 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 Contract N00014-89-J-1988, and by an Office of Naval Research graduate fellowship.

Contents

Chapter 1

Introduction

We are experiencing a dramatic increase in the use of computer communication in both our professional and personal lives. A global electronic infrastructure is fast becoming a reality, and is bringing with it sweeping changes in the way people communicate, do business, and conduct their daily lives. Electronic communication for mail, information exchange, and financial and consumer services are becoming commonplace. And the continued proliferation of distributed computing will undoubtedly inspire new uses for the technology. These applications will bring increased demands for software performance and reliability, as well as an abundance of new software for distributed computing.

Distributed algorithms will be at the heart of this new software. Distributed algorithms are the protocols by which the computers in a distributed system cooperate towards the solution of a problem. Unlike sequential algorithms, used to solve problems on a single processor, 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. Since a given program's computation may unfold nondeterministically, designing and reasoning about distributed algorithms is inherently difficult. Therefore, 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 [47, 48] 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 thesis 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 animating 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.

The Spectrum implementation is faithful to the original I/O automaton model, as presented by Lynch and Tuttle. This model is particularly well suited for describing collections of asynchronous processes that communicate through message passing. However, not all distributed algorithms are best described using private local state and message-passing communication. It is sometimes convenient to describe a distributed algorithm as a collection of processes that communicate through shared variables, or as a collection of system layers, arranged so that each layer makes use of the internal states of lower layers. With a view towards broadening the class of algorithms to which our simulation system is applicable, we present two extensions of the I/O automaton model. The first extension permits automata to communicate through atomic accesses to shared variables. The second extension, called *superposition*, allows programs to be constructed in layers, such that higher layers may observe the internal states of lower layers. We propose new language constructs and simulation system enhancements to support each of

these model extensions.

Another contribution of this thesis is the formulation of a general problem called *logically synchronous multicast* and a highly concurrent protocol to solve it. We show how this protocol could be used in the Spectrum Simulation System to achieve distributed simulation of algorithms expressed as I/O automata.

We now turn to a brief introduction to distributed algorithms. Following this, we present the design goals for the Spectrum Simulation System, and draw distinctions between simulation systems and other sorts of software development tools. The chapter concludes with an overview of the thesis.

1.1 Distributed Algorithms

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:* Choose exactly one distinguished "leader" process (to coordinate some computation).
- *mutual exclusion:* Grant permission for the use of a shared resource (e.g., a printer) in response to user requests so that no two users have permission simultaneously.
- *global snapshot:* Construct a recent consistent picture of the state of the entire system (as in an audit of a distributed banking system).

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. The purpose of this thesis is to demonstrate how this can be achieved. We now elaborate on the design philosophy behind the Spectrum Simulation System.

1.2 Design Goals

We have said that the aim of this work is to construct a research tool for the design and study of distributed algorithms that integrates theoretical modelling techniques with algorithm simulation, visualization, and testing. In this section, we become more specific about this

objective. In designing any software tool, it is important to formulate and adhere to a set of design principles. The philosophy behind the Spectrum simulation system is described by the design principles that follow. We argue generally for the importance of each principle, and describe lower-level design goals that follow naturally from them.

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. Of course, 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. This brings us to the next design principle.

The language should be natural for expressing a large class of distributed algorithms. This design principle has as much to do with the choice of a formal model on which the language is based, as it has to do with the design of particular programming language constructs. The inherent properties of distributed algorithms and systems lead us to the following specific requirements. The model and language should reflect the fact that processes in distributed systems generate outputs autonomously and may receive inputs 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 language and simulation system should encourage experimentation. Often, a researcher does not know exactly where to look for new insights, but discovers them through

a process of exploration and experimentation. It is important that a research tool facilitate this process. This principle implies a number of specific design goals:

1. The write/simulate/modify cycle should be short. That is, the length of time required to modify an algorithm and start the simulation should be small.

2. 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.

3. Logically independent concerns should be orthogonal. 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. Extraneous information for the configuration, visualization, and debugging should not clutter up the algorithm code or interfere with its execution.

4. User effort should be focused on experimenting with *algorithms* rather than finding obscure program errors. This means providing a statically type-checked language with a rich set of built-in data types.

5. Flexible mechanisms should be provided for controlling and studying executions. For example, the system should provide automatic detection of invariant violations, flexible and simple graphical mechanisms for configuring systems and controlling visualization, a choice of scheduling options, and the ability to go backward/forward in an execution and to generate a trace file.

In general, support for experimentation means that it should be easy to modify the algorithm and manipulate the simulation.

Finally, it is important to design for **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. The main goal here is 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 are a concise description of the design philosophy for the Spectrum Simulation System. Their influence is evident in the Spectrum design. We will use these principles in Chapter 6 to evaluate Spectrum and compare it with related languages and systems.

At this point, we should say a few words to distinguish simulation systems from other kinds of software development tools. The purpose of a simulation system, and Spectrum in particular, is to generate executions of algorithms for study and analysis. The difference between a simulation system and an animation system, such as BALSA [10, 11], is subtle but important. In general, the purpose of an animation system is to teach an already well-understood algorithm to others. An animation system typically has two kinds of users, those who set up the animation and those who watch the animation in order to understand the algorithm. Animations are typically rather involved, are constructed by embedding extra procedure calls in the algorithm itself, and are often tailored to a particular algorithm execution or input. In a simulation system like Spectrum, the person setting up the visualization does not necessarily fully understand the algorithm. Since the purpose of the simulation system is to allow an algorithm designer to experiment with the algorithm in order to understand it more fully, we are not interested in fancy animation tricks that require special knowledge of the algorithm executions. We want visualization techniques that are simple enough to be set up quickly, general enough to accommodate any possible execution of the algorithm, and flexible enough to encourage experimentation. Also, as mentioned in our design goals, we want the visualization mechanisms to be clearly separated from the algorithm itself.

A simulation system is also not a theorem prover. A simulation system may be used to assist in program verification by checking properties of particular executions. However, it does not prove properties about all possible executions (as do theorem provers such as LP [21, 22] or Isabelle [52]), and it does not perform exhaustive search to check properties of all possible states (as does the Statemate system [26], which we will discuss later).

One final note of clarification is that we are primarily concerned with the simulation of complex asynchronous algorithms on a sequential machine, with an eye to understanding these algorithms and proving their correctness. A large amount of research has been done in the area of discrete event simulation, where the emphasis is on fast simulation of algorithms with real-time constraints in order to study their time performance. That research has emphasized improving simulation performance through concurrency. (For example, see Misra [49].) However, despite the difference in emphasis, some ideas from discrete event simulation are relevant to this thesis, particularly to Chapter 9 which addresses distributed simulation.

1.3 Thesis Overview

This thesis is divided into two parts. In the first part (Chapters 2 to 6), we present the Spectrum Simulation System, beginning with a review of its theoretical foundations and ending with an evaluation in terms of the design goals we have just stated. Motivated by this evaluation, the second part of the thesis (Chapters 7 to 9) proposes several extensions to the model and system. We now present a detailed overview of both parts of the thesis.

We have said that the Spectrum Simulation System is a research tool for the design and study of distributed algorithms expressed as collections of I/O automata. 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 Chapter 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 Chapter 4, provides facilities for generating executions of these automaton systems. The graphical user interface, described in Chapter 5, is used both for defining the configuration and for controlling the simulation. Spectrum is written entirely in C [34] and runs on DEC Microvax workstations. The user interface is built on top of the X11 window system [56]. In Chapter 6, we conduct an evaluation of Spectrum using the design goals described in Section 1.2. In

that chapter, we draw comparisons with related languages and systems, and reflect upon the experiences of Spectrum users.

Motivated by the evaluation of Spectrum, Chapters 7 and 8 propose two extensions to the I/O automaton model in order to express (and eventually simulate) a wider class of distributed algorithms. The first extension, *shared memory*, allows a collection of automata to make atomic accesses to shared variables. This extension results in a unified model for expressing two large classes of distributed algorithms (message-passing algorithms and shared memory algorithms). A complete assertional proof of Dijkstra's classical shared memory mutual exclusion algorithm [15] is presented to illustrate the shared memory definitions. Another way to model shared state in the I/O automaton model is to model the shared variables as I/O automata that respond to requests to access the variables. We present a general theorem that relates atomically accessed shared memory to the asynchronous invocation-response implementation. The second extension, *superposition*, allows one to describe an algorithm as a series of layers such that higher layers may observe the internal state of lower layers. Besides adding to the expressive power of the Spectrum language, this extension will be particularly useful for monitoring global state invariants during simulation. To illustrate the superposition definitions, the global snapshot algorithm of Chandy and Lamport [13] is presented with a complete proof of correctness. For both model extensions, corresponding language and simulation system extensions are proposed.

Most of the thesis is concerned with simulation of distributed algorithms on a single sequential machine. *Distributing* the simulation, besides being an interesting exercise in itself, can also reduce the simulation time. In Chapter 9, we define the *logically synchronous multicast* problem, which imposes a natural and useful structure on message delivery order in an asynchronous system. In this problem, a computation proceeds by a sequence of *multicasts*, in which a process sends a message to some arbitrary subset of the processes, including itself. A logically synchronous multicast protocol must make it appear to every process as if each multicast occurs simultaneously at all participants of that multicast (sender plus receivers). Furthermore, if a process continually wishes to send a message, it must eventually be permitted to do so. We present a highly concurrent solution to the logically synchronous multicast problem and describe how the logically synchronous multicast protocol can be used to distribute the simulation system. Related broadcast protocols are also discussed.

Chapter 2

The Model

The I/O Automaton model [47, 48] has been chosen as the foundation of the Spectrum Simulation System primarily because it is a natural model for describing distributed algorithms. Careful proofs using this model have been constructed using a variety of techniques for a wide range of algorithms (for examples, see [9, 19, 24, 41, 43, 45, 46, 47, 58, 59]). In this chapter, we present a review of the I/O automaton model adapted from [48]. Interested readers are referred to that paper for more details, motivation, examples, and results. In the course of presenting the model, we highlight those properties that help make the model a solid foundation for a distributed algorithm simulation system. The final sections of this chapter compare the I/O automaton model with related models and justify our selection of the I/O automaton model as formal framework for the Spectrum Simulation System. Further discussion of our choice of this model is contained in Chapter 6.

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]

- $sig(A) = \prod_{i \in I} sig(A_i)$,

[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 $\vec{s}$.

- $states(A) = \prod_{i \in I} states(A_i)$,
- $start(A) = \prod_{i \in I} start(A_i)$,
- $steps(A)$ 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.

We denote the set of fair executions of A by $fairexecs(A)$. We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A, and we denote the set of fair behaviors of A by $fairbehs(A)$. Similarly, β is a *fair schedule* of A if β is the schedule of a fair execution of A, and we denote the set of fair schedules of A by $fairscheds(A)$.

The definitions of composition and fairness imply certain natural relationships between the (fair) executions of a composition and the (fair) executions of the individual components. For

example, the following lemma from [48] states that (fair) executions of component automata can often be pasted together to form a (fair) execution of the composition.

Lemma 2.1: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Suppose α_i is a (fair) execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $acts(A)$ such that $\beta | A_i = sched(\alpha_i)$ for every $i \in \mathcal{I}$. Then there is an (fair) execution α of A such that $\beta = sched(\alpha)$ and $\alpha_i = \alpha | A_i$ for every $i \in \mathcal{I}$. Moreover, the same result holds when *acts* and *sched* are replaced by *ext* and *beh*, respectively.

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.

2.5 Alternative Models

The I/O automaton model is only one of a number of formal models that have been used for reasoning about concurrent systems. A review of alternative models, with an emphasis on techniques for proving algorithm correctness, is contained in [47]. But one's choice of a formal model not only influences the way in which one reasons about algorithms, but also has a strong influence on the way in which one describes algorithms, and particularly the ease with which this is done. Therefore, as we stated in our design goals for the Spectrum Simulation System, it is important to be sure that one chooses a formal model that is natural for expressing the class of algorithms one wishes to describe. In this section, we briefly describe three popular formal models that have been used for describing distributed systems: CSP [31], Unity [14] and Statecharts [27, 28]. In the course of this discussion, we highlight those differences in expressive power that led us to choose the I/O automaton model as the basis of the Spectrum Simulation System. Here, we discuss only the formal models. We will discuss related programming languages and systems in our evaluation of Spectrum in Chapter 6.

2.5.1 CSP

Hoare's *Communicating Sequential Processes* (CSP) [31] is a close relative of the I/O automaton model. A CSP program consists of a set of *processes* written as sequential programs. Each program may contain statements that attempt to send or receive data over *channels* connected to other processes. The channels are synchronous, meaning that the data transfer occurs simultaneously at both ends of the channel, only after both the sender and the receiver are at the appropriate points in their programs. Thus, unlike in the I/O automaton model, a process that is not prepared to receive data may block a process that is prepared to send the data. This

makes CSP unnatural for describing systems in which the individual processes are autonomous.

Many distributed algorithms have the property that different processes in a system may be at completely different points in the execution of their protocols. In such algorithms, a process typically must be able to service requests from other processes (such as a request for a resource) at any time. Thus, describing a component in a distributed system as a single sequential thread of control is rather awkward, since this sequential thread must continually "poll" its incoming channels for such requests. Partially addressing this problem, CSP provides a language construct that allows a process to attempt to send or receive data over multiple channels at a given point in its program; whichever of these data transfers succeeds first is the one executed. This is a powerful construct, but its inherent synchrony does not fit well with the properties of a distributed system. The nondeterministic control flow and input-enabling property of I/O automata combine to provide a more suitable mechanism for expressing this kind of distributed algorithm.

In Chapter 6, we will discuss the Occam programming language [32, 53] based on CSP.

2.5.2 UNITY

Another programming model, UNITY (which stands for Unbounded Nondeterministic Iterative Transformations) [14], abandons the sequential control flow of CSP in favor of nondeterministic choice. A UNITY program consists of a set of *statements* that access a global shared memory. At each step in the (infinite) execution, a statement is selected and executed. Schedules are constrained to be *fair*, meaning that each statement is executed infinitely often. One may think of each statement as a separate process, which is given fair turns to take steps. Since UNITY programs do not terminate, the notion of algorithm termination is defined in terms of a *fixed point* in the execution, after which no statements cause state changes. The UNITY model has a programming logic that is useful for constructing rigorous correctness proofs of algorithms.

To model distributed computation in UNITY, one declares variables that represent channels and writes statements for sending and receiving data that update those variables. Since there is no notion of an "input action" in UNITY, processes must actively read the shared variables in order to become informed of the output of other processes. This rules out synchronous interprocess communication. Modularity is a problem in UNITY because the interfaces between

program modules are not describable in terms of well-defined sets of actions at module boundaries, as in the I/O automaton model, but must be deduced from the program variables that each component accesses. Collections of communicating processes are combined into a single program using UNITY's *union* operator, which, as its name suggests, simply takes the union of the sets of statements of the individual processes to form the new program. One may speak informally of certain UNITY statements as belonging to a particular process, but there is no formal notion of separate system components with their own actions and local variables.

2.5.3 Statecharts

Statecharts [27, 28] provide a modular way to describe complex systems. Essentially, each orthogonal component of a statechart is a finite state machine that, in response to an event, may make a state transition and (optionally) generate a new event. The state of the system is the collection of states of the components.

A complication of the statechart semantics is the notion of a *chain reaction*. Whenever an input occurs, the entire system makes an atomic state transition. But since a given event may be an input to several statechart components, a statechart may generate many new events in response to a single input event. These new events may be inputs to still other system components, and so on. Such a chain reaction is considered to be an atomic step of the system, and the order in which events occur within a chain reaction is important in determining the resulting state of the system. This sort of system behavior does not occur in the I/O automaton model because an atomic step of an I/O automaton may involve a single input action or a single output action, but not both. We will say more about this separation in Chapter 6.

Another important difference between statecharts and I/O automata is that statecharts are finite state machines. This makes them amenable to graphical programming, but results in a loss of expressive power. Since each state of a statechart component is represented explicitly, it is difficult to use statecharts to express many kinds of distributed algorithms (e.g., those that use unbounded counters or message buffers).

The Statemate system [26], based on the Statechart model, provides a graphical editor for building statecharts, a statechart simulator, and automatic translation into Ada and C. Statemate exploits the hierarchical structure of statecharts by permitting users to design and study

complex systems at varying levels of detail. And since statecharts are finite state machines, Statemate can also provide exhaustive testing. We will discuss Statemate further in Chapter 6.

2.6 Summary

In this chapter, we described the I/O automaton model, which has proven to be a useful tool for describing distributed algorithms and proving their correctness. In the following chapters, we will see that many of the same properties that have contributed to the success of the model also provide a good foundation for both a programming language and a simulation system for distributed algorithms.

Chapter 3

The Spectrum Language

The three main components of the Spectrum Simulation System are the programming language, the simulator, and the user interface. In this chapter, we present the Spectrum programming language, whose purpose is to provide a means to express algorithms as *I/O automaton types*, the building blocks of *I/O automaton systems*.

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 [48] 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. In contrast, 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. The language is structured to integrate these five components into an easily digestible form. For example, state components are declared first in an automaton description, actions are clearly marked

as either 'input' or 'output,' steps in the transition relation involving a particular action are defined immediately following the appearance of that action in the signature, and a simple construct is used to clearly divide the output actions into separate classes.

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.

Since the Spectrum language is part of a research tool for algorithm design and study, 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.

In order to shorten the write/simulate/modify cycle, the language is interpreted, provides a convenient set of built-in data types, and is statically type-checked. These last two properties 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 chapter is organized as follows. We begin in Section 3.1 by identifying

those 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 [39]. Finally, Section 3.5 describes special language support for verification, analysis, and visualization. A grammar for the language syntax and a list of built-in functions are contained in the appendix. This chapter contains sufficient detail to serve as an introduction to the Spectrum programming language for potential Spectrum users. The implementation of the language is described in the next chapter.

3.1 Theoretical Foundations

In the previous chapter, we reviewed the I/O automaton model on which the Spectrum programming language is based. Before presenting the language, we identify those features of the model that provide a solid foundation for a useful and implementable programming language, and also discuss those features of the model that must be modified slightly in order to produce an implementable language. We begin with the model features that can be captured directly in the language:

- Automata 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.[1]
- Input actions are always enabled, and output actions are under the control of only one automaton. In this way, we can describe systems of autonomous components.
- Atomicity is made explicit. I/O Automata are described in terms of state transitions, where each transition is executed atomically. There is no doubt about which are the atomic steps.

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

- Communication is accomplished by shared actions. Since shared actions take place simultaneously at all the participating automata, it is easy to reason about communication among the automata.

- Automata may be composed to construct complex systems. This gives rise to flexible modularity and straightforward encapsulation mechanisms. System modules can be composed in a variety of ways (for example, in a hierarchy or a series of layers). Reasoning about modules is accomplished in terms of the actions that occur at their boundaries.

- Transition relations are nondeterministic. 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.

- The model is simple, having relatively few concepts. Many possible language features, such as elaborate control constructs and user-defined abstract data types, are orthogonal to the basic concepts of the model and could be added to the Spectrum language without disruption.

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:

- 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.

- 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. Luckily, 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. *All* of these executions are supposed

to be correct, as opposed to the usual automata-theoretic case where only one need be correct. Therefore, substituting randomization for nondeterminism is an acceptable design decision. 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. Alternatives to randomization in the scheduler, such as deterministic schedulers and user-controlled scheduling, will be discussed later.

- The model assumes that systems may have infinitely many automata and that the automata all exist at the beginning of an execution. Our implementation can support only a finite number of automata, all in existence at the beginning. In addition, each automaton is allowed only a finite number of equivalence classes in the partition.[2]

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. Then it is the responsibility of the simulator to guarantee fairness. (As we will see in the next chapter, the current implementation of Spectrum provides two schedulers, a round-robin scheduler and a randomized scheduler. The round-robin scheduler guarantees that all executions are fair, while the randomized scheduler produces fair executions with high probability.)

Having discussed generally how the properties of the I/O automaton model can be captured in an implementable language, we now turn to the details of the Spectrum programming language design.

[2] An interesting possibility for further work would be to add language constructs for creating new automata and classes dynamically to provide the illusion of an infinite set.

3.2 A Separation of Concerns: Automaton Types vs. Configurations

At the beginning of this chapter, we said that 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.

One may 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. Therefore, it is important to have an understanding of what is contained in a configuration in order to understand the programming language fully.

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). At his or her option, the user may assign each instance a string name, and may specify a set of directed edges that organizes the instances into an arbitrary graph. As part of the configuration process, 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.

To avoid confusion, we emphasize that the programming language itself is not used to create

composed automaton types, but only to define the lowest level automaton types in a system. In Chapter 5, we will say more about creating composed types, and about the configuration process in general.

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. Later in this chapter, we will become more specific about how configuration data are accessed within automaton type definitions.

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 are presented one by one, and are followed in Section 3.4 by an illustrating example. Since defining an I/O automaton system configuration is not strictly part of the programming language, we defer that discussion to Chapter 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. In this section, we describe Spectrum's built-in data types, mechanisms for constructing complex data types, and functions for manipulating values of various data types. In later sections, we will see how these data types are used in automaton type definitions.

The language supports 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. Constructed types may be arbitrarily nested. For example, one can define sets of tuples. For

programming convenience, any data type may be assigned a mnemonic name using a DATA declaration. In the following example, the first line defines the type *buffer* to be a multiset of tuples, where each tuple has a string and an automaton_id. The second line defines the type *messages* to be a mapping from automaton id's to buffers.

```
DATA buffer    multiset(tuple(msg:string, to:automaton_id))
DATA messages  mapping(automaton_id, buffer)
```

Our notion of type equality is structural equality, where names of tuple fields are considered to be part of the structure of a type. For example, the type *same_type* below is equivalent to *buffer* above, but *different_type* is not. User-defined names for constructed data types are simply conveniences in writing programs; they are irrelevant in type-checking. One may think of a DATA declaration as a macro definition. Recursive declarations are not permitted.

```
DATA pair           tuple(msg:string,  to:automaton_id)
DATA same_type      multiset(pair)
DATA different_type multiset(tuple(name:string, id:automaton_id))
```

Every data type, including all constructed types, has operations for comparison ($=, <, \leq, >, \geq$) and assignment. In addition, each data type has its own special set of operations. The data type of the return value for each operation is inferred (statically) from the data types of its arguments. For example, if `myset` is declared to be a set of integers, then `set_minimum(myset)` returns a value of type integer. Similarly, the expected data types of different arguments of a given function are checked (statically) for compatibility. For example, if `mytuple` is a tuple variable, then the expression `set_insert(myset,mytuple)` would generate an error at load-time, since one cannot insert a tuple into a set of integers. Operations (in addition to assignment and comparison) for the various data types are summarized below and listed in detail in Appendix B.

- integers: Operations include arithmetic inverse, addition, subtraction, multiplication, truncated division, and mod. Additionally, integer variables may be used in "summary mappings" for algorithm visualization purposes. We will say more about this in Chapter 5.
- booleans: Operations include testing for truth, negation, and logical and, or, xor, and implication. Like integer variables, boolean variables may be used for summary mappings.

All predicates in the language are boolean expressions. These include action preconditions, tests in conditionals, and state invariants. We will discuss all of these in detail later in this chapter.

- automaton id's: Operations for this data type 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. Later, we will see how configuration data may be used to define the signature and transition relation of an automaton instance.

- strings: One may manipulate strings using generic assignment and comparison operators. Also, a decimal number represented as a character string may be converted to an integer. This is particularly useful when one wishes to assign numerical names to automaton instances in the configuration, and then perform arithmetic operations on those names.

- tuples: Using record notation (to an arbitrary depth), one can reference the fields of tuples by name, and then operate on them individually according to their particular data types. Tuples are convenient for storing multipart messages. For example, a message's destination, text string, and sequence number can be manipulated as a unit using a tuple.

- sets and multisets: Operations include initialization (to empty), creation of a singleton set, boolean test for empty set, finding the size of a set, testing for set membership, and set (or multiset) union, intersection, and difference. Furthermore, one may select the minimum or maximum element from a set, select an element at random, or select an element (or maximal subset) such that a given boolean expression is satisfied. Quantification over a set (forall, exists), and iteration over a set (forall x in set s do ...) is also supported.

In distributed algorithms, sets and multisets are commonly used as buffers (e.g., to keep track of pending messages).

- sequences: Operations include initialization (to the empty sequence), inserting at the front or back, deleting from the front or back, deleting the first occurrence of a specific element, test for membership, and finding a random element without modifying the sequence. Sequences are particularly useful for implementing stacks and queues, as well as maintaining history information.

- mappings: Operations include initialization (in which a default value is specified for "unmapped" elements of the domain), assignment of the value of a mapping for a particular element of the domain, and evaluation of the mapping at a particular element of the domain. Mappings are useful for representing dynamically changing functions (e.g., to keep track of status information for each adjacent edge). In addition, we will see that mappings are useful data types for algorithm visualization.

In order to prevent references to undefined variables, each base type in the language has a default value. Similarly, each type constructor has a default value, defined recursively on the basis of the defaults for its component types. As we will see in Section 3.3.3, a mechanism is provided for explicitly defining the initial values of variables, but these defaults are used when no explicit initial value is provided.

Readers should refer to Appendix B for a complete summary of the supported operations. 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

We use action types to 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`. This is described further in Section 3.3.3.

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 every action in the system is under the control of a single automaton. (In the current implementation, `a.owner` is not accessible in automaton type definitions, but can be duplicated, of course, as part of the user-defined argument of an action.)

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. In the next section, we describe how the states, signatures, and transition relations of automaton types are defined.

3.3.3 Automaton Types

As mentioned earlier, 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 use the graphical interface to compose several automaton types into a new "composed" automaton

type. In this section, we consider only the first method; the latter is discussed when we describe the user interface in Chapter 5.

Every automaton type declaration begins with a type name. For example, the line

```
AUTOMATON channel
```

says that we are about to define an automaton type with the type name *channel*. Each subsequent line (up to the next automaton declaration) is used to define the various pieces of the channel automaton type. Recall that an I/O automaton consists of a set of states, a set of initial states, a signature, a transition relation, and a partition of the locally controlled actions. We now present the programming language constructs that Spectrum provides for defining each of these pieces of an automaton type.

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.)

State components are private storage. 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. An automaton's state may be referenced almost anywhere in the definition of the transition

relation, but may be modified only in *effect* clauses. We will describe this in more detail when we discuss transition relations.

The initial values for state components are defined in a special input action called *initially*, which occurs as the first action of every execution. We will see an example of this in Section 3.4. As we described in Section 3.3.1, any state component not explicitly assigned an initial value is given a default value, but it is considered good style to initialize all state components.

Signatures

The action signature of an automaton type consists of a set of input actions and a set of output actions.[3] 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 he set of neighbors of the automaton in the configuration. (Recall that `a.to` refers to the "to" component of the action argument.)

[3] In the I/O automaton model, the signature of an automaton consists of a set of actions, divided into input actions, output actions, and internal actions. For simplicity, we have restricted the Spectrum language to input and output actions only. An internal action can simply be regarded as an output action appearing in the signature of only one automaton.

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 avoid this computational disaster 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. If a weight is not explicitly assigned, a default weight of 1 is used.[4]

3.4 Example

As an 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 [39]. 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 3-1. 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 3-2. 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.

[4] It has been suggested that this default be changed to 100 in future versions of the system.

```
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 3-1: 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 3-2: A parameterized class.

3.5 Support for Verification, Analysis, and Visualization

In the previous sections, 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 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. As we will see in Chapter 4, after each step of the automaton's execution, the set of invariants is checked, and the execution is interrupted if an invariant is violated.

In assertional proofs of distributed algorithms, it is quite common to write invariants that involve the states of many different system components (i.e., *global invariants*). Unfortunately, the INVARIANT construct in Spectrum allows one to express invariants only on the local state of an automaton. We discuss this problem further in Chapter 6 and propose a solution in Chapter 8.

3.5.2 Spectators

Assertional proofs use invariants on the state of a computation as a means to show properties of the *behavior* of an algorithm. That is, our primary 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 Chapter 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 $sig(\mathrm{M})$ as follows:

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

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 $\mathrm{UserTry}_i$ and $\mathrm{UserExit}_i$, and has inputs Crit_i and Rem_i. A $\mathrm{UserTry}_i$ 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 $\mathrm{UserExit}_i$ 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 $sig(\text{M})$. 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, Crit_i,.... This says, for example, that a process will not issue a try request while in its critical section.

We define the set $scheds(\text{M})$, the allowable external behaviors of M, as follows. Let β be a sequence of actions in $sig(\text{M})$. Then $\beta \in scheds(M)$ 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. One may notice that this schedule module specifies only safety properties; it does not require that any progress be made. In Chapter 7, we will see a similar schedule module that specifies both safety and liveness properties.

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 3-3 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

```
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 3-3: A spectator for mutual exclusion.

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 $ok = true$ is violated if and only if no UserExit occurred since the 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* [37] and the ALWAYS construct of UNITY [14], which we discuss in Section 6.1.2. 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. The above is a relatively simple example, but a pseudovariable can be used to summarize 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 Chapter 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.

The MAINTAIN clause is executed after the effects clause of each action and before any local invariants are checked. The expressions of the MAINTAIN clause are executed sequentially.

3.6 Summary

In this chapter, we saw that a separation of I/O automaton types and the configuration of an I/O automaton system is a central part of the Spectrum design. We presented constructs for defining the states, signatures, transition relations, and partitions of automaton types and described linguistic support for verification, analysis, and visualization. As a partial summary

clause	state	action arguments	class parameters	configuration data
STATE	declared	—	—	—
INVARIANT	read	—	—	read
MAINTAIN	read/modified	—	—	read
ACTION	—	declared	—	—
CLASS	—	—	declared	read
WHERE	—	read	read	read
PRE	read	—	read	read
SEL	read	read/modified	read	read
EFF	read/modified	read	read	read

Figure 3-4: Data that may be declared, read, or modified by the various clause types.

of the constructs provided in the Spectrum programming language, Figure 3-4 lists, for each kind of clause, those categories of values that may be declared, read, or modified by that clause.

In Chapter 6, we present an evaluation of the Spectrum Simulation System in terms of the design goals we set out in Chapter 1. Although the programming language will be discussed in this evaluation, at this point we say a few words about the programming language in isolation. We should emphasize that the semantics of the language are entirely faithful to the I/O automaton model, with the exception that nondeterminism is replaced by randomization. Since a large class of distributed algorithms is expressible in the I/O automaton model, the language is natural for expressing a large class of distributed algorithms. In addition, I/O automaton composition provides us with the modularity necessary to enable users to write spectator automata to check and analyze algorithm executions in such a way that they do not interfere with the executions themselves.

The Spectrum language provides constructs for writing well organized descriptions of algorithms as I/O automaton types. The names of automaton types, their state definitions, signatures, transition relations, and classes are all laid out in a natural way in automaton type definitions. Extra information, such as invariants for program verification and pseudovariables for program visualization, are separated from the other constructs, so as not to obscure the algorithm itself, yet are written using the same language mechanisms. The language provides a rich set of built-in data types with associated operations that conveniently match those mathematical objects typically used to express distributed algorithms. However, language mechanisms

for creating user-defined data types would be useful.

The applicative style of the Spectrum syntax (i.e., the lack of infix operators) makes it difficult to "think in Spectrum." Spectrum users have found it beneficial to first express algorithms using a higher-level notation similar to that used by Lynch and Tuttle. Once the algorithms are expressed in this way, writing the automaton type definitions in the Spectrum language is straightforward. For example, Gupta [25] presents a number of distributed algorithms written both in the higher-level notation and in the Spectrum programming language. In order to avoid this extra step in the programming process, it would be useful to have a "sugared" version of the syntax that permits infix operators, and an accompanying preprocessor to translate the sugared version into the syntax we have presented here. Such a preprocessor might also have an option to generate a "publication version" of the language in a form that could be used by a text processor.

In Spectrum, control flow is determined completely by preconditions on actions and the random choices of the scheduler. However, because people tend to think sequentially, the individual processes in distributed algorithms sometimes take a sequential form. Therefore, it may be useful to add syntactic sugar for simple control flow constructs that could be translated into the language presented here. Additional state components would be added to the automata by the preprocessor to keep track of the "program counter," and preconditions would reference this additional information.

Chapter 4

The Spectrum Simulator

In the previous chapter, we presented the Spectrum programming language used to define I/O automaton types, the building blocks of I/O automaton systems. In this chapter, 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 purpose of this chapter is to impart an understanding of the organization of the simulator, and of how the simulator interacts with the other components of the Spectrum system. Such an understanding is helpful not only for programmers interested in extending the capabilities of Spectrum, but also for Spectrum users interested in learning how the simulator generates executions of their automaton systems. It is also instructive to see how the simple semantics of the Spectrum programming language (and the I/O automaton model) facilitate a clean simulator implementation.

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. In this chapter, we concentrate on the functionality and implementation of the simulator. Since all user interaction with the simulator is mediated by the user interface, we leave that discussion for the next chapter.

We organize our presentation of the simulator around its four main logical components, the *loader*, the *interpreter*, the *execution loop*, and the *scheduler*. 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. The job of the *interpreter* is to evaluate expressions represented as automaton types data structures. These expressions may be used to evaluate preconditions to determine the set of classes with enabled actions, determine the argument values for actions, determine the set of participants in a given action, effect the state transitions of automata, and check state invariants. The interpreter contains implementations for all of the built-in operations for the data types in the language. The interpreter is called on to evaluate expressions at the discretion of the *execution loop*. The execution loop controls the entire sequence of events that take place in an execution, from initialization to termination detection. In short, the execution loop manages the simulation of each event in an execution. 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, according to the selected scheduling algorithm. We now present each of the logical components of the simulator in detail.

4.1 The Loader

As we have said, the job of the loader is to parse an automaton types file and construct the data structures that represent the types in that file. It parses data type declarations, action type definitions, and automaton type definitions, and checks for syntax errors and type errors in all three. All of this is accomplished in a single pass through the automaton types file.

Upon parsing a data type definition, the loader builds a data structure to represent that type. In the case of DATA declarations, the loader keeps a *data types table* that associates data type names with pointers to their corresponding data typo structures. Data types structures are used by the loader not only to represent declared data types, but also to represent automaton

state definitions, data types for the return values of functions, the data types of class parameters, and the data types of quantified variables. These data structures are used in the loader for type checking, and are used extensively by the interpreter in order to determine the proper treatment of arguments to generic functions.

Just as the loader creates a *data types table* to associate data type names with data type structures, it also creates an *action types table* that associates each action type name with a pointer to a data structure representing its argument type. For each action type, the action types table also holds a pointer to a list of all the automaton types having that action type among its set of input actions. This list is used during simulation to help determine the set of automata that participate in an event of that type.

The main job of the loader is to parse each automaton type definition and build a data structure to represent it. Again, the loader creates a table, called the *automaton types table*, with an entry for each automaton type containing: the name of the automaton type, the type of its state, a list of its input actions, a list of its classes, and two lists of expressions corresponding to the MAINTAIN and INVARIANT clauses of an automaton types definition. As one would expect, an automaton type's state definition is is represented as any other data type structure. Similar data structures are used to represent the signature, classes, and transition relation of an automaton type.

The loader checks for syntax errors and performs type checking on all expressions. When a function expression is parsed, the loader creates a data type structure representing the type of the return value of the function, determined for polymorphic functions 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. In addition to the above checks, the loader enforces restrictions on the various clause types, as shown in Figure 3-4. For example, attempting to assign to a state component in a precondition causes an error.[1] In general, the loader generates helpful error messages when it encounters syntax errors (such as undefined variables, functions or tuple components, and extra or missing arguments).

[1]There is one exception to the rules listed in Figure 3-4: In order to permit certain visualization functions, the loader *does* allow the configuration data to be modified in certain special cases. For further information, see the description of the configuration data functions in the appendix.

4.2 The Interpreter

The interpreter provides 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[2] 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. This is one place where the simplicity of the Spectrum language (and the I/O automaton model) contribute to a clean simulator implementation: all of the above functionality essentially boils down to evaluating expressions. In order to evaluate expressions, the interpreter creates and manipulates data structures representing values of expressions, state components, and action arguments. These data structures are similar to those used to represent data types in the loader, but contain the actual values instead of data type information. 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.

As we mentioned above, all the functionality of the interpreter is provided by expression evaluation. In order to determine whether an action of a given type is enabled in the current state of an automaton, one calls on the interpreter to evaluate the precondition for that action. Similarly, to assign to the arguments of an action, one asks the interpreter to evaluate the selection clause for that action. In order to determine the set of participant automata for an action, one asks the interpreter to check the WHERE clause of that action for each automaton instance having that action type in its input signature. Finally, to make state transitions, one asks the interpreter to evaluate the EFF clause of each participant. All such calls to the interpreter are under the control of the execution loop, presented next.

4.3 The Execution Loop

The execution loop is responsible for controlling the entire simulation. Upon invocation, the execution loop assumes that the automaton types data structures and configuration data struc-

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

tures are already in place. Its first task in starting up a simulation is to initialize the data structures for each of the automaton instances.[3]

Recall that each automaton instance in a configuration has an associated automaton type. Using the definition of this automaton type, the simulator creates a data structure for each automaton instance in the configuration. This data structure contains: a list of the classes of that instance, the current state of that instance, and a list of checkpointed states (used for rolling back the simulation). It should be emphasized that *each instance* has its own list of classes, initialized according to the corresponding automaton type definition. Each class contains the values of its parameters and a bit indicating whether or not it contains an action that is currently enabled. For each class in the type definition, if the class is defined without parameters, then a single corresponding class is created for the instance. However, if the class of the automaton type is parameterized, then the interpreter is called on to determine the set of possible values for each parameter according to the set expression associated with that parameter and the configuration data for that automaton instance; then a separate class is created for the instance for each possible combination of parameter values. When created, the enabled bit of each class is set to 0 (false).

After all the classes have been created, the interpreter is called 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 instance is found to be enabled, then the enabled bit of that class is set to 1 (true). The scheduler, to be discussed in the next section, is informed of each class so enabled. This completes the initialization procedure.

A high-level description of the execution loop is shown in Figure 4-1. 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

[3]The user interface reserves space in the configuration data structure for a single pointer to be used for this purpose by the simulator.

```
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 4-1: Pseudocode for the execution loop.

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 evaluated to determine the action argument. 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, presented in the next chapter, is informed of both the action and the set of participants in order to update the display.[4] The simulator also provides, for the user interface, functions that textually format the data types of the states of automaton types, the values of the states of automaton instances, and the current action name with its argument value. In addition, the simulator provides a function that extracts the values of selected state components of automaton instances in order that the user interface may

[4] 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, etc.

update the graphics display to reflect automaton state changes.

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. If the trace file option is selected, at the end of each iteration of the execution loop, the current step number, action name, and selected state information are written to a file. In the next chapter, we will see how the user specifies what state information is written to the file. In order to achieve more efficient rollback of the simulation, the user may specify an autosave interval k, indicating that the state of the automaton instances and the scheduler should be checkpointed after every k steps of the execution. When the simulator is requested to roll back (or forward) to step number n, the simulator reverts to the last checkpointed state prior to step n and then advances the simulation to that step.

4.4 The Scheduler

It is the job of the scheduler to maintain a list of all classes containing enabled actions, and to choose the next class to take a step at each iteration of the execution loop. 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 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 [49]), 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 event may become disabled by the occurrence of other events.

4.5 Summary

In this chapter, we described the four main components of the Spectrum simulator: the loader, the interpreter, the execution loop, and the scheduler. The simulator precisely implements the semantics of the Spectrum programming language, and therefore captures the semantics of the I/O automaton model.

The simulator is extensible in several directions. One way to exploit the modularity of the I/O automaton model would be to define built-in automaton types that are used by programs in the same way as user-defined automaton types, but are actually treated differently by the system in order to perform low-level system functions. For example, one might have a built-in automaton called a file manager that has input actions `OPEN(fn,read/write,id)`, `GET-STRING(fd,len)`, `PUT-STRING(fd,s,len)`, and `CLOSE(fd)`, where `fn` is a file name, `read/write` is a boolean value, `id` is an automaton id, `fd` is a file descriptor for an open file, `len` is an integer length, and `s` is a string, and output actions `RETURN(v,id)` where `v` may be a file descriptor, a character, an OK signal, or an error signal. The actions would have the usual file operation semantics, and one could write a schedule module specification for this automaton for use by programmers. However, under the covers, the internal implementation of the automaton by the simulation system would be different. Rather than executing as an I/O automaton, input actions of the file manager would cause commands to be sent to a separate UNIX process to execute the corresponding system call. Upon completion of the system call, the appropriate output action of the file manager would become "enabled." One might imagine similar built-in automata to manage other system services such as print queues or even network

connections, allowing the interaction of Spectrum programs on several machines.

One might also extend the simulator by adding a variety of schedulers. For example, the deterministic variety might include, in addition to a round robin scheduler, a least-recently-executed scheduler that gives the next turn to the class that has been waiting the longest to take a step. The randomized variety might include, in addition to the current randomized scheduler, those that dynamically change the weights of classes in order to produce "strange" executions, possibly taking into account the number of steps that a class has been waiting to take a step. A more radical extension, which would involve language extensions as well, would be to allow programmers to write their own scheduling algorithm, or to provide "hints" to the scheduler. Such hints might be generated by a spectator automaton, taking into account history information. Such algorithm-specific scheduling could be used in an adversarial way, to try to generate executions that are incorrect or have poor performance. In Chapter 6, we will consider still other possibilities for improvements in the scheduling mechanism, including user intervention and a real-time scheduler.

As another improvement to the simulator, we suggest an optimization for updating the set of enabled classes at each iteration through the execution loop. Since action preconditions typically do not involve all the state components of an automaton, and since each state transition typically does not change all the state components of an automaton, we suggest the following. Let the loader mark each precondition with a concise representation (a bit vector, say) of the state components on which it depends. Then, when an automaton takes a step, compare the state components that have changed to this bit vector for each precondition, and only reevaluate those preconditions that depend upon changed state components. This optimization requires a small amount of additional storage to hold the current values of the preconditions for each automaton, but, depending on the algorithm being simulated, could result in a significant savings in simulation time.

In this discussion, we have identified some possibilities for extensions and improvements to the simulator that are largely independent of the language and the user interface. In Chapter 6, we will discuss possible extensions to the Spectrum Simulation System that involve all three system components.

Chapter 5

The Spectrum User Interface

This chapter describes the user interface of the Spectrum Simulation System. The user interface has two purposes. It 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. 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. 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. Also, one may undo and redo modifications to the configuration. An important configuration option is the ability to build new automaton types by composing others. All of the above editing options are available for creating composed types, as well.

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 actions occur in the system, participating automata are highlighted and state changes are displayed using color and text. In order to provide flexible exploration of algorithms, the choice of state components to be represented as colors may be changed during simulation. Also, one can view the simulation at various levels of detail by selectively opening up windows onto the automaton instances at

any level in the composition hierarchy. 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 [34] on top of the X11 window system [56] and requires four color planes. The current implementation runs on DEC Microvaxes. A three-button mouse is used for most interface commands.

5.1 Overview of the Spectrum Interface

The *main window* of the Spectrum interface consists of, from top to bottom, a *banner*, a *pulldown menu bar*, a *configuration area*, a set of three *menus*, and a *message area*. The left edge of the banner shows the name of the automaton types file currently loaded into the simulation system, and the right edge of the banner shows the name of the configuration file that was last loaded.

The pulldown menu bar contains three entries, "SET UP," "CONFIGURE," and "SIMULATE." At all times, one of the latter two entries is highlighted to indicate whether the interface is in configure mode or simulate mode. The *set up menu* is used for file management, and the *simulate menu* for selecting among various simulation options, such as the choice of a scheduler. The *configure menu* is intended to be used for special-purpose configuration options, such as enforcing acylicity in the instance graph when this is a desired topology assumption. However, the configure menu is unused in the current implementation. The largest area of the main window, the configuration area, shows the configuration currently loaded into the interface. *Undo* and *redo* buttons are available at the upper left of the configuration area.

The three rectangular areas near the bottom of the screen are, clockwise, the *types menu*, the *edit menu*, and the *color spectrum*. These are explained later. The message area across the bottom of the main window is used for providing textual information to the user, and for keyboard interaction.

In addition to the main window, various *auxiliary windows* may be created. In configure mode, auxiliary windows are used to create composed types and to set up algorithm visualization. In simulate mode, they are used to display automaton state information. We will describe

these in detail later in the chapter.

5.2 Configure Mode

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, a set of adjacent edges in a directed graph, and a parent in the composition hierarchy. In configure mode, the Spectrum interface provides tools for creating and editing configurations graphically. 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. 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. The remainder of this section describes the options available in configure mode.

5.2.1 The Types Menu

In the Spectrum interface, automaton types are represented graphically as different shapes. The *types menu* near the bottom of the main window contains a row of icons, each available to represent a different automaton type. At any time, exactly one entry of the types menu is *selected*, shown as a filled polygon. The selected entry is used for the "create" and "change type" edit options (see Section 5.2.3). One can assign a name to each icon in the types menu; if an icon's name matches an automaton type definition in the currently loaded automaton types file, then that icon is used to represent automata of that type.

Any entry in the types menu may be "opened," causing the creation of a new window. The upper right of the new window contains the icon and name of the opened type. If the name of the menu entry matches the name of an automaton type defined in the currently loaded types file, then the new window displays, in formatted text, the data type of the state of that automaton type. Otherwise, the new window is used for creating (or modifying) a composed type (see Section 5.2.5). In either case, the new window may be used to define summary mappings (see Section 5.2.6).

5.2.2 The Edit Menu

Almost all modifications of the configuration are accomplished using the options in the *edit menu*, located near the bottom right of the main window. Using the edit menu options, one can create automaton instances, delete or move them, or change their types. In addition, one can use the connect and disconnect options to create and delete directed edges between automaton instances. Edit options are useful not only for specifying a configuration, but also for creating natural spatial arrangements of the automaton instances so that simulations may be more easily comprehended. At any given time, exactly one entry in the edit menu is selected; this entry is highlighted in a special color.

Since an edit option remains selected after its use, one may repeatedly use an edit option without returning the mouse to the edit menu. For example, one can delete many automaton instances by simply selecting *delete*, and then choosing each instance to be deleted. The edit menu options are discussed further in the following sections.

5.2.3 Automaton Instances

To create an automaton instances of a given type, one selects the *create* option in the edit menu and the desired automaton type in the types menu, and then positions the desired instances in the configuration area. To delete automaton instances, one selects the *delete* edit option and chooses the instances to be deleted. The edit options for moving and changing the type of an instance work similarly.

When an automaton instance is created, it is assigned a unique automaton identifier, which appears below the icon as its name. One may rename an automaton instance , but changing the name of an automaton does not change its unique identifier. Both the system-assigned identifier and the user-defined name are accessible in automaton type definitions (see Chapter 3).

If an automaton is an instance of a composed type (see Section 5.2.5), it may be "opened" in configure mode. One may rename the components of an instance of a composed type. However, no other modifications of instance components are permitted, in order to ensure consistency among all automata of the same type. For example, one cannot create a new component in just one instance of a composed type.

5.2.4 Configuration Edges

Automaton instances may be connected in a directed graph. 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 (see Chapter 3).

To place edges between pairs of automaton instances, one selects the *connect* edit option and selects the pair of automaton instances for each desired edge. A directed edge is formed from the first instance in each pair to the second. To delete an edge, one selects the *disconnect* edit option and selects the two endpoints in the same way.

5.2.5 Creating Composed Types

It is often desirable to define an automaton type that is the composition of several other automaton types. 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.

To create a composed type, one assigns the desired type name to an unused shape in the types menu and opens that type, as described in Section 5.2.1. The new window created may be used just like the configuration area in the main window. To create the composed type described in the preceding paragraph, one would use the *create* edit option to create one "instance" of the user process automaton type and one "instance" of the message buffer automaton type inside the window for the composed type. Each instance of the composed type is then the composition of instances of those two automaton types.

All of the edit menu options are available for creating composed types. For example, one may create a composed type containing several "instances" of another type arranged in a particular directed graph. 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.

5.2.6 Summary Mappings

During simulation, colors are used to display the values of important components of an automaton's state. One might say that certain state components summarizing the state of the automaton are "mapped" to colors of the icon representing the automaton. These summary mappings may be established at configuration time, but may also be changed during simulation. 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. Both integer and boolean state components may be mapped as colors. The MAINTAIN clause (see Chapter 3) is 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. This tends to make the visualization easier to understand. When setting up summary mappings for a composed type, one may choose the middle or border of any component as the state component to be mapped out. For example, if A is the composition of B and C, one may wish to map the middle of B to the middle of A, and map the border of C to the border of A. Then, during simulation, whatever state component is mapped to the border of C will also map to the border of A.

5.2.7 Undo and Redo

All of the functions provided in the edit menu, plus changes to summary mappings, may be "undone" by using the undo button in the window where the modification was performed. For example, to undo a modification to a composed type, one uses the undo button in the window for that type. The undo feature is "infinite," meaning that one may undo changes all the way back to the beginning of the current session. (A new session begins when the interface is started and whenever a configuration file is loaded.) A "redo" option is available for redoing modifications just undone.

5.2.8 Saving and Loading Files

The setup menu option *load types* is used to load an automaton types file written in the Spectrum programming language. The simulator does parsing and type-checking, and sets up the appropriate data structures for the interpreter. If syntax errors, type errors, or other problems occur while reading the file, an error message appears at the bottom of the main window and a description of the errors is output by the simulator. (See Chapter 4 for details.)

The *save configuration* option in the setup menu is used to save a configuration in a file for a later session. The entire configuration, including composed types and summary mappings, is saved in the file named. The *load configuration* option is used to load a previously saved configuration. On loading a configuration, the interface checks that the summary mappings in the configuration file properly correspond to the automaton types currently loaded. For example, it checks that if the first state component is used in the summary mapping, then that component has type integer or boolean. Each summary mapping that does not match up properly is discarded. Since these compatibility checks are made only when a configuration file is loaded (and not when an automaton types file is loaded), one must load the types file before creating a configuration or loading a configuration file.

The fact that automaton types files and configuration files are saved independently means that a given types file can be used with many configuration files, or many types files with a given configuration file. When the latter arrangement is used, one may need to change the summary mappings for different types files when the automaton state components have different types.

5.2.9 Data Structures

In configure mode, the user interface builds and maintains data structures to represent the system configuration. Its main data structure is a table indexed by automaton id that holds the configuration data relevant to that automaton (its type, its user-supplied name, its parent in the composition hierarchy, a list of the automaton id's of its components (if it is an instance of a composed type), and lists of its incoming and outgoing edges), as well as information relevant to the display of the icon representing that automaton (the id for the X window containing the icon, the icon's color(s) and position in that window, and the id for the associated auxiliary window if one has been created). In addition, for each automaton instance, space (for a single

pointer) is reserved for the simulator to associate state and class information with each instance. Similar information is kept for automaton types, and in particular for composed types. In simulate mode, configuration information (including the pointer reserved for the simulator) is made available to the simulator through a narrow interface of procedure calls and macros.

5.3 Simulate Mode

Simulate mode is used for running simulations. In this section, we describe how the interface is used to invoke the various features of the simulator. For details on the simulator itself, see Chapter 4. We begin by discussing the options available in the simulate menu, and then describe what happens in the interface during a running simulation.

5.3.1 The Simulate Menu

With the exception of the scheduling options, all options in the simulate menu are available in simulate mode. (Since it is not permissible to change schedulers in the middle of a simulation, the scheduler must be selected prior to entering simulate mode.) The following options are available in the simulate menu.

Simulation Control Options: If *single step* is selected, the user advances the simulation one step at a time. If *continuous* is selected, the simulation runs to completion or until it is interrupted by the user or by an invariant violation.

Scheduling Options: The choices are *randomized* or *round robin*. See Chapter 4 for a description of these schedulers.

Set Pause: Using this option, one can specify a pause (in seconds) between the steps of a continuous simulation.

Set Skip: This option is used to instruct the interface to update the display only after every n steps, where n is the skip value chosen.

Autosave: The user may set an autosave value of n, which causes the simulator to checkpoint the state of the simulation every n steps. A low (non-zero) value of n makes undo and

redo faster, but slows down the simulation (see Section 5.3.4). If the autosave value is zero, then autosave is turned off.

Trace: This option may be used to specify a trace file. See Section 5.3.6 for a description of what is written to a trace file.

5.3.2 The Running Simulation

On entering simulate mode, the automaton instances are initialized and the simulation is paused just before the first step. As the simulation runs, the display is updated as follows. After each step of the execution, provided that the simulator is in single-step mode or that the pause value is at least one second, 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 the message area. In addition, the colors of the automaton instance icons are updated according to the summary mappings, and any open state windows are updated. The icon colors and state value displays are updated in both single-step and continuous mode, regardless of the pause value. However, when the skip value is nonzero, all display updates take place only after the specified number of steps. Similarly, no display updates take place for the intermediate states that occur while the simulation is being advanced or undone to a particular step number, but the display is updated once the desired state has been reached. Whenever the simulation is paused, it is possible to:

- resume the simulation,
- abort the simulation and return to configure mode,
- select options from the simulate menu (except changing the scheduling option),
- open entries in the types menu to change the summary mappings,
- open automaton instances to view their components (in the case of compositions) or to view the values of their state variables,
- roll back the simulation to some earlier step number, or
- advance the simulation to some later step number.

We have already discussed the simulate menu options (Section 5.3.1) and summary mappings (Section 5.2.6). In the following sections, we discuss viewing automaton instance state variables, rolling back and advancing the simulation, error messages, and trace file.

5.3.3 State Windows

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. During simulation, any automaton instance may be opened. If the automaton is an instance of a composed type, then the auxiliary window created contains its 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 the window created, called a *state window*, contains a textual display of the values of the state variables. As described earlier, all state information (whether graphical or textual) is updated as the simulation runs.

5.3.4 Execution Rollback

In understanding an algorithm, it is often useful to observe the events that led up to a particular state. Therefore, the simulator provides the ability to roll back the simulation to a particular step. Like all simulator functions, the rollback/advance option is controlled with the user interface, which allows the user to select a particular step number to return to (or advance to). To support this feature, the simulator checkpoints the states of the automata, the scheduler, and the seed of the random number generator at the beginning of the execution and periodically according to the interval selected with the *autosave* option in the simulate menu. If the checkpoints occur frequently, rollback will be fast. On the other hand, very frequent checkpointing wastes memory space and slows down the simulation. One may change the checkpoint interval whenever the simulation is paused.

5.3.5 Error Messages During Simulation

Two different types of error messages may occur during simulation. Both are reported on the Unix standard error file and in the trace file (if one is open). The first type of error is program

error, such as an attempt to select a random element from an empty set, or an attempt to divide by zero. The second type of error is the violation of a specified invariant of an automaton. In either case, the automaton name and id at which the error occurred, along with the action name and arguments, are reported with the error message. Violation of an invariant causes the simulation to become paused so that the user may examine the state of the system and roll back the simulation in order to discover the source of the error.

5.3.6 Trace Files

If a trace file is specified, the following information is appended to it by the simulator. At the start of the simulation, the seed for the random number generator is recorded in the trace file so that the execution may be reproduced later. After each step, the step number and the action name (with its arguments) are written. Whenever a state window is opened or updated, the values of the state variables of that automaton are recorded in the trace file. In this way, the states of the automata of interest are recorded automatically without cluttering the trace file with extra state information. All error messages (both program errors and invariant violations) are written to the trace file. Finally, whenever the execution is rolled back or advanced to some particular step number, that information is also recorded.

5.4 Summary

The Spectrum user interface provides features for constructing system configurations, creating summary mappings, and controlling and observing algorithm executions. An important purpose of the Spectrum user interface is to provide support for experimentation with algorithms. Configuration files are created, edited, and saved independently of automaton types files so that one may use many different configurations with a given set of automaton types, and vice versa. Also, the summary mappings used for algorithm visualization are created easily and may be changed easily during algorithm simulation for maximum flexibility in observing algorithm executions. In addition, the composition hierarchy is exploited to allow users to observe algorithms at various levels of detail.

In Chapter 6, we will discuss a number of ways in which Spectrum's support for experimen-

tation could be enhanced. In addition, Gupta [25] makes a number of specific suggestions for improving user interaction in Spectrum.

Chapter 6

System Evaluation

In the previous chapters, we presented the three main components of the Spectrum Simulation System: the programming language for defining automaton types, the simulator for generating executions of automaton systems, and the user interface for building automaton system configurations and controlling simulation and visualization. In each chapter, we pointed out features of the design that were motivated by the design goals outlined in Chapter 1.

In this chapter, we use these design goals to evaluate the system as a whole. First, in Section 6.1, we compare Spectrum with a number of related languages and simulation systems. Then, in Section 6.2, we conduct a thorough evaluation of Spectrum in terms of the design goals, taking into account the comparisons drawn in Section 6.1, as well as our own experience using Spectrum and the experiences of other 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 evaluation will motivate the research presented in the remaining chapters of the thesis.

6.1 Comparisons with Related Systems

In this section, we survey several languages and systems that are based on formal models and 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. The languages and systems we discuss are Occam [32, 53], Unity [14], Statemate [26], and DEVS [60].

6.1.1 Occam

The Occam programming language [32, 53] is based on Hoare's *Communicating Sequential Processes* (CSP) [31], discussed briefly in Chapter 2. In Occam, each process has a sequential program, which may attempt to send or receive data over *channels* connected to other processes. A process may wait to send or receive data at multiple channels at a time. The channels are synchronous, meaning that the data transfer occurs simultaneously at both ends of the channel, only after both the sender and the receiver are at appropriate points in their programs. (That is, inputs are not always enabled.) Thus, a process that is not prepared to receive data may *block* a process that is prepared to send the data. This makes Occam rather unnatural for describing distributed algorithms in which the individual processes are meant to act autonomously. Also, Occam does not have a general handshake mechanism, but instead limits synchronous communication to pairs of processes. Therefore, it is impossible to write an atomic broadcast as a primitive operation in Occam. Pairwise communication also prevents one from constructing monitoring devices, like Spectrum's spectator automata, to monitor interprocess communication in Occam. However, other languages based on CSP do permit such monitoring (for example, see [55]), but one must be careful to ensure that the monitoring process does not interfere with the computation by blocking events from occurring.

The sequential process control flow provided by Occam is convenient for describing algorithms that are inherently sequential. However, it can be cumbersome for describing distributed algorithms in which a given process may interact with other processes at different stages of the protocol. This is because one must continually wait on multiple channels, and then act according to the type of message received. In Spectrum, a more general control flow mechanism is provided. The state changes required for each kind of input action are expressed in isolation, and arbitrary predicates on the state (instead of sequential control flow constructs) are used to determine whether or not each output action is enabled individually. For convenience in describing algorithms in which each process thread is sequential, it might be interesting to design a sugared version of the Spectrum language that would build sequential control flow constructs

on top of the precondition mechanism. A preprocessor for this language could introduce extra control variables (program counters) into the automaton state that would be used as preconditions on locally controlled actions. Such control flow constructs would be used for describing the activities of each class of an automaton, and input actions would be unaffected.

The system configuration in Occam is described within the code of the algorithm. This results in a poor separation of the algorithm from the system configuration, but does allow for dynamic process creation, useful for describing some important kinds of distributed systems. We will return to dynamic process creation later in the chapter.

6.1.2 UNITY

The UNITY programming model was described briefly in Chapter 2. Recall that a UNITY program consists of a set of *statements* that access a global shared memory. At each step in the (infinite) execution, a statement is selected and executed, such that each program statement is executed infinitely often. We pointed out that to describe message-passing communication in UNITY, one declares variables (say, queues) to represent the channels, and writes statements to modify those variables for sending messages (enqueue) and receiving messages (dequeue). But since there is no notion of an "input action" in UNITY, receivers must keep polling the "channels" with dequeue statements in order to become informed of the output of senders. This is similar to the problem we encountered in Occam, where a receiver must continually wait on all of its input channels, except that in UNITY the sending process would not be blocked by the receiver. In Spectrum, it is easier to express message-passing computation primarily because the underlying model is so well suited for that purpose.

As far as mechanical checking of algorithm executions is concerned, it would not make sense to consider writing spectator-like devices in the UNITY programming language. This is because the entire state is shared, and there is no notion of a shared action. However, it would be quite sensible to monitor invariants on the system's global state.

We pointed out in Chapter 2 that modularity is a problem in UNITY because the interfaces between program modules are not describable in terms of well-defined sets of actions, but only in terms of the program variables that they access. The closest approximation to composition in the UNITY model is the *union* operator, which simply takes the union of the sets of statements

of the individual modules to form one large program. One could use this operator to build up complex programs and experiment with the programs by substituting modules as in Spectrum. However, the value of doing this in UNITY is limited because the distinctions between components become blurred in the final program; the components all share a single state space and do not have their own action signatures. For example, several different components might have the same assignment statement; in the union of those modules, one would not be able to tell which component was "responsible." In fact, it does not even make sense to speak of this in UNITY because the identity of an individual component is lost when the union operator is used.

Another operator provided in UNITY for combining programs is the *superposition* operator. This is useful for constructing layered systems in which higher layers may observe (but not modify) the variables used by the lower layers. We will return to superposition later in this chapter, and again in Chapter 8.

Two similarities between UNITY programs and Spectrum automaton type definitions are worth mentioning. There is an INITIALLY section in a UNITY program that defines the initial values of variables. It is written as a set of equations, where each variable appears at most once on the left hand side of an assignment. It must be possible to compile the equations into a sequence of assignment statements such that all the variables become initialized. This bears some resemblance to Spectrum's *initially* action, except that the initially action is written just as any other input action: it is not constrained to initialize all the variables, and is allowed to assign variables multiple times in the course of performing the initialization. UNITY has an ALWAYS section whose purpose is to maintain pseudovariables, like Spectrum's MAINTAIN clause. Again, the ALWAYS section is written as a set of equations, while the MAINTAIN clause is a sequence of assignments. A further restriction on the ALWAYS section is that a variable assigned in that section must not appear on the left hand side of an assignment in any other part of the program. This restriction guarantees that any equation appearing in the ALWAYS section is an invariant on the program state.

6.1.3 Statemate

The Statemate system [26] is based on the Statechart model [27, 28] described in Chapter 2. Recall that Statecharts provide synchronous nonblocking multi-party communication, but allow a process to receive input events and generate output events in one atomic action; this results in atomic "chain reactions" that can be difficult to reason about. Statechart programs are described as hierarchical finite state machines. Unfortunately, the finiteness rules out a large class of distributed algorithms.

Statemate provides a graphical editor for building statecharts, a statechart simulator, and automatic translation into Ada and C. Since statecharts are finite state machines, Statemate can also provide exhaustive testing. Statecharts do not separate the algorithm description from the system configuration; they are one and the same. However, it is relatively easy to add components to Statecharts in the Statemate system, and the hierarchical structure of Statecharts allows one to observe system executions at varying levels of detail.

Since a statechart describes a system that makes an atomic state transition in response to each event from the environment (and does not take steps on its own), it is difficult to use a statechart to model systems in which there are autonomous processes that may generate outputs at any time. For example, consider a sender process that makes a request to a data link process to transmit packets of data. One would like to separate the request event that is output by the sender process from the packet sending event that is output by the data link process. However, if the only actions from the environment are the requests from the sender process, then there is no way (in a statechart) to model a delay between the sender's request and the corresponding sending event output by the data link layer; they must be treated as a single atomic step. With I/O automata, one would model the request as an input action to the data link process and the send as an output action of that process. This seems to be a more realistic view of what actually occurs in such a system, and allows one to reason about what might happen between the two events. For example, one might wish to study executions in which the requests come in spurts at a rate faster than the packet sends go out. One could simulate such a delay in the Statemate system by having the environment generate a separate event (like a clock tick) that would result in the corresponding data link send, but this clutters up the external interface and puts the environment in control of a delay that is logically internal

to the system.

In addition, since each state is drawn explicitly, large data structures or variables with large domains (such as message buffers or counters) are placed either in the environment or the Statemate *database*, where they are not visible as part of the program state. For example, a timer cannot be modelled explicitly in the system. One can write start-timer and stop-timer events that act as signals to some device in the environment of the statechart, and write a timeout event that comes from the environment when the time has expired. But a more natural approach is to model the timer as an explicit system component that has start-timer and stop-timer as inputs, takes internal steps to decrement the counter, and issues timeout as an output. This appears not to be possible with Statecharts, but is easy to accomplish using I/O automata. In general, it is often convenient to model distributed algorithms as closed systems of autonomous processes, but this is not possible using the Statemate approach.

One main advantage of the Statemate system is the graphical programming approach. In Spectrum, there are graphical configuration and visualization tools, but the language is textual. An interesting possibility for further work would be to design a graphical programming language for a restricted class of I/O automata. Such a language might not support the infinite state capability of I/O automata, but could allow system components to take steps autonomously.

6.1.4 DEVS

DEVS [60] is an object-oriented system in which system components (called *models*) have input and output ports that may be *coupled* in a hierarchical fashion. Unlike Occam, DEVS does allow a given output to be directed to multiple destinations. However, in any port-based interconnection mechanism, an output cannot be transparently directed to different destinations depending on its *value*. In Spectrum, action names are used (instead of ports) to relate the outputs of one module with the inputs of others. Spectrum's is a more general mechanism. For example, in Spectrum an output may be directed to different destinations depending on its value. One could attempt this in DEVS by creating a different port for each destination, but this would require that the module producing the output know the relationship between values and destinations. To avoid requiring that the output module have this knowledge, one could create a separate port for each possible value, but this would create an overwhelming

configuration task.

As in any discrete event simulation system, scheduling in the DEVS simulator is based on global time. This allows one to describe and analyze real-time systems, as well as study the time performance of ordinary asynchronous algorithms. However, the algorithms to be simulated must explicitly manipulate times to keep track of process step time, even if their actions do not depend on the amount of time elapsed. One would like a better separation of the algorithm from the timing concerns. That is, rather than cluttering up the algorithms with timing details, one would like to specify timing constraints on the processes of the system, and then allow the simulator to handle the rest "under the covers." Later in this chapter, we will discuss such an approach for extending Spectrum for the study of real-time systems.

6.2 Design Goals Revisited

Having compared Spectrum with a number of other languages and systems, we now evaluate the system in terms of the design principles outlined in Chapter 1. Recall that the design principles were as follows:

- The design must be faithful to a formal model.
- The language should be natural for expressing a large class of distributed algorithms.
- The system should encourage experimentation with algorithms.
- The design should achieve economy and integration.

We now consider Spectrum in terms of each of these principles and the corresponding subgoals identified in Chapter 1.

6.2.1 Spectrum and the I/O Automaton 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. Finally, the distinction between internal and output actions in the I/O automaton model was dropped in the Spectrum design. Each of these design decisions was mentioned in earlier chapters, but we now consider them and their implications in more detail.

The nondeterminism present in the I/O automaton model is not of the usual complexity-theoretic flavor. That is, we do not speak of I/O automata "guessing" solutions to problems and then verifying the answers deterministically. And we do not speak of an input as being "accepted" by an I/O automaton if there exists some computation path that leads to an "accept state." Instead, nondeterminism is used in order to make algorithms and their correctness proofs more *general*. That is, 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. Therefore, the decision to substitute randomization for nondeterminism is a legitimate one. 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, random executions are produced according to some probability distribution by a particular random number generator. So, "strange" executions, such as those that might lead to starvation of a contender in a mutual exclusion algorithm, are unlikely to arise in executions generated by Spectrum, unless one deliberately skews the probability distribution by placing appropriate relative weights on the automaton classes. One might consider adding mechanisms to the simulation system, such as user-defined schedulers or dynamically changing class weights, in order to generate a wider variety of system executions. Still, 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. The limitation of a static finite collection of automata (and a finite number of classes in the partition) 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 [41, 43] and [40].) 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 decision to omit internal actions from the Spectrum language was made only in order to simplify the initial design. Adding internal actions to the language would be a simple matter. An interesting related extension would be to support a hiding operator in the user interface. Such an operator would be useful additional support for viewing algorithm executions at various levels of detail, and also for restricting trace files to contain only the actions of interest.

6.2.2 Expressive Power

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 Chapter 7, we present extensions to the I/O automaton model for describing shared memory algorithms and propose related extensions to the Spectrum Simulation System.

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 [14]. 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 Chapter 8, we extend the I/O automaton model to permit superposition of program modules and propose related extensions to the Spectrum Simulation System.

The expressive power of Spectrum for describing distributed algorithms based on message passing is quite good. However, as we mentioned in Section 3.6, some Spectrum users (particularly those without prior experience with I/O automata) find that it is difficult to "think" in the Spectrum language. They find that it is easier to first write algorithms using a higher-level approach, and then translate them into the Spectrum language. This is related to the issue of expressive power, but has more to do with ease of programming. It appears that most of the problem is due to the syntax of the language, which lacks infix operators. A sugared version of the language with infix operators might allow programmers to write their algorithms directly in the language. Additional language features that may help here are the control flow constructs

discussed in Section 6.1.1 and a syntax for assigning to all the components of a tuple at once (for example, `s.tuple = <s.comp1,s.comp2>`).

Other mechanisms that would add useful expressive power to the language include additional built-in data types and new operations for the existing ones, as well as the ability to create user-defined data types with their own operations. In addition, it would be useful to access systems services, such as file management and device I/O from within automata. An interesting way to accomplish this would be to treat each system service as an automaton that is built into the system, as described in Section 4.5.

6.2.3 Experimentation

Since Spectrum was designed as a research tool, a number of design decisions were made in order to provide support for experimentation with algorithms. The first step in providing support for experimentation is to make it easy for the user to concentrate on the relevant details, and help the user to avoid wasting time. The fact that systems are configured graphically and their executions are observed visually saves time and effort in setting up and analyzing simulations. Also, since the Spectrum language is statically type-checked, the loader can detect the "silly mistakes" in automaton type definitions so that users do not waste time searching for obscure program errors. The rich set of built-in data types and the fact that the language is interpreted also serve to shorten the write/simulate/modify cycle. But besides helping to save the user time in writing algorithms and setting up and running simulations, Spectrum provides a number of flexible mechanisms that directly address the problem of experimenting with algorithms. We now review some of these.

An important part of the Spectrum design was the decision to separate the automaton types and the system configuration. As a result of this decision, Spectrum users can experiment with a given algorithm in a variety of system configurations. Crucial to making this 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. At a more fundamental level, without the

parameterization mechanism, there would be no way to break symmetry among automata of the same type.

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. Visualization is useful for debugging algorithms, understanding how they work, and studying their efficiency. In general, there are two ways to accomplish visualization: declarative and imperative [54]. In imperative visualization, one embeds procedure calls in the algorithm being studied in order to effect changes in the display. At any time in the simulation, the image on the display is a product of the history of these procedure calls. This allows one to construct elaborate program animations that are rather difficult to set up and modify. In contrast, the declarative approach established relationships between state information of the algorithm with points on the display. With this approach, the animations tend to be simpler, but it is easy to set them up and to modify them quickly. Furthermore, with the declarative approach it is easy to to update the display when one rolls back the simulation, since one only need be concerned with the current state and not the entire history of the execution. Summary mappings use the declarative approach. As a result, it is possible to set up algorithm visualizations quickly in Spectrum, and to modify them easily, even during simulation.[1] In addition, when one wishes to create special variables to be used in summary mappings, the MAINTAIN clause allows one to keep those variables up to date without obscuring the rest of the program code. We also take advantage of I/O automaton composition in the summary mappings mechanism so that users may observe the states of algorithm executions at varying levels of detail.

The modularity provided through I/O automaton composition is also useful for experimenting with algorithms. 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.

[1] Currently, it is possible to color the edges of the configuration graph as part of the visualization capabilities of the system. However, this mechanism is imperative and such visualizations are harder to change. One approach to making the edge colors imperative would be to associate an automaton with each edge in a configuration and then use summary mappings.

An important part of experimenting with algorithm is generating many different executions for them. In Spectrum, there are currently two scheduling algorithms, randomized and round robin, as described in Chapter 4. The randomized scheduler allows one to assign weights to the classes in order to simulate some processes running faster than others. One question is whether these weights should really be part of the automaton type definitions, as they are now, or part of the configuration. Making them part of the configuration would allow more flexibility in scheduling, for one might be permitted to change the weights during simulation, just as it is possible to change summary mappings as the simulation proceeds. For this change, the user interface might be modified so that each automaton instance had an associated "pop-up" menu containing a list of its classes and their associated weights, which could be changed at the keyboard. As a further extension, one might highlight the set of classes containing enabled actions in the menu, and allow the user to specify, at any point during simulation, which class should take the next step. Upon selecting a class, the user might be presented with another pop-up menu containing the set of enabled actions in that class and be permitted to choose an action (and possibly select its arguments). Alternatively, 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, but this option could lead to chaos in an algorithm unless used conservatively. At the very least, it would be appropriate to check any declared invariants on the state of an automaton whenever the user makes changes.

Chapter 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 invariant violations, or violations of the problem specification, or for computing performance statistics (such as the number of messages generated in an execution). An important point to remember is that 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

we have noted that 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 invariants on the private states of individual components, but also about invariants that involve the states of many system components. In the current Spectrum implementation, such invariants cannot be checked. One might imagine adding a special language feature to permit writing such global invariants. However, the superposition extensions we propose in Chapter 8 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 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. This is because Spectrum does not support a notion of time. However, several researchers have already extended the I/O automaton model for the study of real-time systems, and have used the extended model to construct timing-based proofs for distributed algorithms [2, 3, 42, 50]. Their work could form the basis of useful extensions to the Spectrum system. Using their approach, timing information in Spectrum would not be manipulated explicitly by the algorithm being simulated (as was seen in DEVS), but 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. In Chapter 4, we suggested some possible optimizations to the Spectrum simulator. But another way to speed up simulations is to introduce concurrency in the simulator. In Chapter 9, we will study the problem of achieving highly concurrent distributed simulation of I/O automata.

6.2.4 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.

6.3 Summary

In Chapter 1, we stated that an aim of this thesis 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 ease 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. Further integrating the tasks of developing algorithms and proving their correctness is likely to be a fruitful area of research. For example, proof techniques, such as *possibilities mappings*, are likely to translate into powerful algorithm development tools.

In this chapter, we compared the Spectrum Simulation System with a number of related languages and systems, evaluated Spectrum in terms of its design goals, and described many possible directions for further work. In the remainder of this thesis, we will consider three of these directions. In Chapter 7, we describe extensions to the I/O automaton model and Spectrum that expand their expressive power by allowing one to describe distributed algorithms that make use of atomically accessed shared variables. In Chapter 8, we describe another extension to the model and system that permits one to superpose I/O automata in order to describe algorithms in terms of system layers. Finally, in Chapter 9, we address the problem of distributed simulation of I/O automata.

Chapter 7

Shared Memory

In Chapter 1 we said that reasoning about algorithms for asynchronous concurrent systems is difficult, primarily because of the arbitrary interleaving of process steps that may occur in an execution, and that researchers have turned to formal models in order to define problems precisely, give unambiguous descriptions of algorithms, and construct careful proofs for safety and progress properties. Formal models allow one to be explicit about the possible interleavings that may occur in a distributed system and may specify which of those interleavings are to be considered "fair" to the individual system components. However, any particular formal model is not necessarily well suited to describing all kinds of distributed algorithms. For example, CSP [31] is best at modelling systems in which components communicate by sending messages over synchronous channels, and UNITY [14] is good at describing algorithms in which components communicate by reading and modifying shared variables.

We have seen that the I/O automaton model is particularly well suited for modelling distributed algorithms described using message passing. Recall that the I/O automaton model is a (not necessarily finite) state machine model that provides extra support for classifying actions as input or output and for describing fairness conditions. Precise problem statements are defined in terms of the input and output actions that occur at the boundary between the algorithm and its "environment." These problem statements may include nontrivial liveness constraints on the behavior of the algorithm. Careful algorithm descriptions are constructed

Sections 7.1 and 7.2 are joint work with Nancy Lynch. Section 7.3 is joint work with Kathy Yelick.

by specifying the states and transition relations of I/O automata. A range of proof techniques, from simple assertional reasoning to hierarchical possibilities mappings, may be used to verify an algorithm satisfies a problem statement. In addition, the model can be used for carrying out complexity analysis and for proving impossibility results. The communication mechanism in a distributed system is modeled as an explicit I/O automaton that shares actions with the other system components. Therefore, the model can accommodate a variety of message passing systems, from systems with strictly FIFO message delivery to those in which messages may be delivered out of order or not at all.

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. Therefore, it would appear that introducing a shared memory mechanism into the I/O automaton model would be a useful unification of these two approaches. The shared memory model of Lynch and Fischer [44] introduced the separation of input and output actions, and was a precursor of the current I/O automaton model. However, until now it has not been clear how to integrate the two basic approaches.

In this chapter, we extend the I/O automaton model to allow modelling of shared memory systems, as well as systems that have both shared memory and shared action communication. A full range of types of atomic accesses to shared memory is allowed, from basic reads and writes to atomic read-modify-write. We define a special class of actions, called "shared memory actions," to model atomic accesses to shared memory. Each shared memory action contains extra information that corresponds to the contents of the shared memory before and after the action occurs. A "shared memory automaton" is then defined to be an I/O automaton that satisfies certain natural conditions regarding its shared memory actions. For example, one condition captures the idea that an access to shared memory must be prepared to observe any value in the memory.

Since shared memory automata are simply special cases of I/O automata, all the I/O automaton model definitions and properties (notably composition and fairness) apply to shared memory automata as well. We show that composing a collection of shared memory automata

(for a given set of shared variables) yields another shared memory automaton (for the same set of variables). To combine shared memory automata having different (not necessarily disjoint) sets of shared variables, we define an "augmentation" operator that is used to expand the set of shared variables for each component before composing. We show that the natural compositionality results hold when we combine shared memory automata in this way. For example, projecting the execution of a composition on the individual components yields executions of those components. Since we expose the observed state of shared memory in the behavior of an automaton, we also achieve compositionality of the *behaviors* of shared memory automata. That is, in the standard sense of I/O automaton composition, the behaviors of a composition of shared memory automata are the same as the composition of the behaviors of the individual automata.

Shared memory automata operate in a system in which the environment is free to change the contents of the shared memory at any time. We define a "closeout" operator, which takes a shared memory automaton and a set of variables and produces a new shared memory automaton in which the given set of variables is made private, absorbed into the local state. In this way, we restrict the set of components in a system that may access portions of the shared memory.[1] We provide an analogous closeout operator on sets of behaviors, and we show that the behaviors of a *closed out automaton* are the same as the *closed out behaviors* of the original automaton.

Just as does the original I/O automaton model, our extended model supports careful problem specification (including both safety and progress properties), unambiguous system description, verification and analysis. Both safety and progress properties of algorithms may be shown using standard proof techniques (e.g., invariant assertions and variant functions). To illustrate these techniques, we present and prove the correctness of Dijkstra's classical shared memory mutual exclusion algorithm using the shared memory I/O automaton model.

Two features of the model will be especially useful in our shared memory extensions: the separation of inputs and outputs and the definition of fairness. The separation of input and output actions will be important for two reasons. First, the fact that each action is under the control of exactly one component means that by simply using actions to model updates to the

[1]The ability to closeout with respect to a subset of the shared variables (as opposed to the entire set) may be likened to lexical scoping of variable declarations in a conventional programming language.

shared memory, we capture the notion of a single module making an atomic update to shared memory (without any active participation by other modules). Second, the fact that input actions are always enabled means that we can use shared memory input actions to construct modules that passively observe the shared memory accesses by others without interfering. We will return to these points in Section 7.1.7. In our example progress proof of Dijkstra's mutual exclusion algorithm, we will rely on the built-in fairness feature of the I/O automaton model in order to reason about progress in a system containing several active, non-failing processes accessing passive shared memory.

Algorithms described using atomic accesses to shared memory are easy to reason about, since one is not concerned with the possible interleavings of invocations and responses for object access. However, the invocation-response approach, in which the invocation of an operation on an object and the corresponding response are modelled as separate atomic steps, fits more naturally with multiprocessor architectures and often supports greater concurrency. Thus, an important problem in multiprocessor algorithm design is to carefully construct the processes and objects in such a way that any the invocation-response system "simulates" an atomic access system. One approach to this problem has been advanced by Herlihy and Wing [30]. They propose a property of objects, called *linearizability*, that permits one to construct invocation-response systems, and then to reason about only those executions in which each response immediately follows the corresponding invocation. We exercise the unified model resulting from our shared memory extensions by showing a formal relationship between the invocation-response approach and the atomic access approach in the context of linearizable objects.

After presenting these results, we suggest language and system extensions for incorporating the shared memory extensions (specifically, the closeout operator) into the Spectrum Simulation System. These extensions would allow simulation of message-passing algorithms, shared memory algorithms, and hybrid algorithms all within a single formal framework. This is an added benefit of building a powerful unified model that accommodates both message passing and shared memory communication.

The remainder of the chapter is organized as follows. In Section 7.1, we define our extensions for shared memory and show some important properties that follow from these definitions. Then, in Section 7.2, we use extended model to present and prove correct Dijkstra's shared

memory mutual exclusion algorithm. Next, in Section 7.3, we use the extended model to establish a formal relationship between the invocation-response approach and the atomic access approach in the context of linearizable objects. Finally, in Section 7.4, we propose extensions to the Spectrum Simulation System that correspond to the shared memory extensions of the I/O automaton model. These system extensions would permit describing and simulating shared memory algorithms.

7.1 Shared Memory Definitions

In this section, we present a set of definitions that extends the I/O automaton model in order to allow modelling shared memory algorithms. *We do not redefine any concepts,* but simply add new concepts to the existing model. We model each system component that accesses shared memory as a restricted I/O automaton called a "shared memory automaton". The fact that shared memory automata are simply special cases of I/O automata means that all the standard definitions and properties of I/O automata (e.g., composition and fairness) can be used directly in descriptions and proofs of shared memory algorithms.

7.1.1 Variables

We will model shared memory in terms of a collection of variables, so the first step is to define what is meant by a variable. We define a *variable* x to have a domain *dom(x)* of values and an initial value *init(x)* $\in$ *dom(x)*. Given a set X of variables, we model a state of X by an *assignment mapping* for X, denoted f_X, that maps each variable $x \in X$ to a value in *dom(x)*. We let F_X denote the set of all possible assignment mappings for X. We define *init(X)* to be the assignment mapping $f_X \in F_X$ such that $\forall x \in X, f_X(x) = init(x)$. If X and Y are sets of variables such that $Y \subseteq X$, we define $f_X|Y$ to be the assignment mapping $f_Y \in F_Y$ such that for all $y \in Y$, $f_Y(y) = f_X(y)$. If X and Y are disjoint sets of variables, and S_X, S_Y are sets of assignment mappings for X and Y, respectively, then we define $S_X \oplus S_Y$ to be the set of assignment mappings S for $X \cup Y$ such that for all $s \in S, s|X \in S_X$ and $s|Y \in S_Y$. As shorthand, we may represent a singleton set of assignment mappings by its only element. For example, if f_X is an assignment mapping for X, we write $f_X \oplus S_Y$ instead of $\{f_X\} \oplus S_Y$. Analogously, for

$f_X \in F_X$ and $f_Y \in F_Y$, we let $f_X \oplus f_Y$ represent its only element when it is clear from context that a mapping (rather than a set of mappings) is called for. If $f \in F_X$, $x \in X$, and $v \in dom(x)$, we define $f_{[x=v]}$ to be the assignment mapping f' such that $f'|(X \setminus \{x\}) = f|(X \setminus \{x\})$ and $f'(x) = v$.

7.1.2 Shared Memory Actions

Since the only "sharing" that occurs in the I/O automaton model is the sharing of actions, we construct shared memory on top of the existing shared action mechanism. We begin by defining a special type of action called a "shared memory action" that will be used to model accesses to the shared variables[2].

We fix $\mathcal{L}$, a universal set of *access labels*. Let X be a set of variables. We define a *shared memory action for* X to be a triple of the form (f'_X, a, f_X), where $f'_X, f_X \in F_X$ and $a \in \mathcal{L}$.[3] We let *sm-acts(X)* denote the set of all possible shared memory actions for X. We say that π is a *shared memory action* iff it is a shared memory action for some X. We say σ is a shared memory step (for X) iff its contained action is a shared memory action (for X).

To construct signatures for shared memory automata, we need the following technical definition. Let Π be a set of actions and X a set of variables. We say that Π is *complete* for X iff $\forall \pi \in \Pi$, if $\pi = (f'_X, a, f_X)$ is a shared memory action for X, then $\forall \hat{f'_X}, \hat{f_X} \in F_X, (\hat{f'_X}, a, \hat{f_X}) \in \Pi$.

Let X and Y be sets of variables such that $Y \subseteq X$. If $\pi = (f'_X, a, f_X)$ is a shared memory action for X, we define its projection on Y, denoted $\pi|Y$, to be $(f'_X|Y, a, f_X|Y)$, a shared memory action for Y. If β is a sequence of actions, all of whose shared memory actions are shared memory actions for X, then we define $\beta|Y$ to be the sequence that results from replacing each shared memory action of β by its projection on Y. Projections on sets of shared memory actions, signatures containing shared memory actions, and sets of sequences containing shared memory actions are defined analogously. If $\sigma = (s', \pi, s)$ is a step with π a shared memory action for X, then $\sigma|Y$ is defined to be $(s', \pi|Y, s)$.

---

[2] In some sense, this is the reverse of what is often done to incorporate message passing into a shared memory model. In UNITY [14], for example, shared queue variables are declared to model "channels" and atomic accesses to these shared queues model "sending" and "receiving" data across the channels.

[3] These triples are action names, not to be confused with the steps of an automaton.

7.1.3 Shared Memory Automata

Let X be a set of variables, and let A be an I/O automaton all of whose shared memory actions are external shared memory actions for X. Let *shared-in*(A) denote the set of shared memory actions that are inputs to A, and let *shared-out*(A) denote the shared memory actions that are outputs of A. We say that A is a *shared memory automaton for* X iff it satisfies the following conditions:

1. The sets of actions *shared-in*(A) and *shared-out*(A) are each complete for X.
2. For all steps $(s', (f'_X, a, f_X), s) \in steps(A)$,
   if $(f'_X, a, f_X) \in$ *shared-out*(A), then for all $\hat{f}'_X \in F_X$, there exists a state $\hat{s}$ and some $\hat{f}_X \in F_X$ such that $(s', (\hat{f}'_X, a, \hat{f}_X), \hat{s}) \in steps(A)$.
3. In the equivalence relation $part(A)$, any two output actions (f'_X, a, f_X) and $(\hat{f}'_X, a, \hat{f}_X)$ are elements of the same equivalence class.

The first condition says that if A has a shared memory action with a given label a, then it has all possible shared memory actions with label a. For input actions, this means that A must be prepared to handle any value it may observe in the shared variables (since inputs are always enabled). For output actions, this condition is simply a technical restriction that makes composition of shared memory automata work out properly, as we will see later. The condition also makes describing the signatures of shared memory automata more convenient, since we need not list all the allowable values of the shared variables for each shared memory action label used.

The second condition says that for each shared memory output step, there exists a step from the same state for each possible assignment of the shared variables. In essence, this says that the preconditions of an output action may not depend on the values of the shared variables. This corresponds with the notion that one cannot observe the values of shared variables except by accessing them, and that one must be prepared to handle any value that might be observed.

The third condition says that the equivalence class membership of an output action may not depend upon the values of the external variables. This is a technical condition that prevents a nonsensical situation in which executions must be "fair" to the different values of the shared variables.

Since a shared memory automaton is an I/O automaton, all the standard I/O automaton definitions for executions, schedules, behaviors, composition, and fairness carry over to shared memory automata.

Theorem 7.1: The composition of a strongly compatible collection of shared memory automata for X is a shared memory automaton for X.

Proof: We know that the composition of a strongly compatible collection of I/O automata is an I/O automaton. Furthermore, since external actions of the components are external actions of the composition, we know that all of the shared memory actions are external actions in the composition. All of these are shared memory actions for X. It remains to be shown that the composition satisfies the three conditions imposed on shared memory automata for X. Condition 1 holds, since the union of complete sets of actions is clearly a complete set. For condition 2, we note that composition does not introduce any new output actions, nor does it remove any existing output actions. Furthermore, input-enabling and the definition of composition imply that for each output step (s'_i, π, s_i) of a component $\mathcal{A}_i$, for all states s' of the composition $\mathcal{A}$, if $s'|\mathcal{A}_i = s'_i$, then there exists a state s of $\mathcal{A}$ such that (s', π, s) is a step of $\mathcal{A}$. Thus, Condition 2 holds. Since the equivalence relation of the composition is the union of the individual equivalence relations of the components, any two actions in the same equivalence class in a component are in the same equivalence class in the composition. Since the set of shared memory output actions for each component is complete, strong compatibility assures us that no two shared memory output actions with the same label occur in different classes of the composition. This guarantees Condition 3. ∎

So far, we have given a general set of definitions for modelling collections of modules that access shared memory. Our accesses allow a module to atomically read the entire contents of memory, perform some local computation (possibly resulting in a state transition), and update the entire contents of shared memory. This general type of shared memory access is, of course, an expensive operation to implement. Therefore, we would like to define systems in which the shared memory accesses are more restricted. For example, in the most restricted case, we might only allow read or write accesses to single shared variables.

Let A be a shared memory automaton for X, let a be an access label of A, and let $x \in X$. We say that a is a

1. *read access to* x iff $\forall (s', (f', a, f), s) \in steps(A)$,

   (a) $f = f'$ and

   (b) $\forall \hat{f} \in F_X$ such that $\hat{f}(x) = f'(x)$, $(s', (\hat{f}, a, \hat{f}), s) \in steps(A)$.

2. *write access to* x *with value* v iff $\forall (s', (f', a, f), s) \in steps(A)$,

   (a) $f = f'_{[x=v]}$ and

   (b) $\forall \hat{f} \in F_X$, $(s', (\hat{f}, a, \hat{f}_{[x=v]}), s) \in steps(A)$.

In a read access to x, the shared memory is unmodified and the new state of A depends only upon the value observed in variable x. In a write access to x, the "before" and "after" states of shared memory differ only in the value of variable x, and the new state of A and the new value of x are independent of the "before" state of shared memory.

We now define a restricted class of shared memory automata called "single-variable read-write automata." In such automata, each access label for a shared memory output is constrained to be a read access or a write access to a single variable. Let A be a shared memory automaton for X, and let ψ be a partition of the access labels for actions in *shared-out*(A) such that there exist exactly two classes in ψ for each variable in $x \in X$, denoted $\psi_r(x)$ and $\psi_w(x)$. The partition ψ is called the *access partition* of A. We say that A is a *single-variable read-write automaton under* ψ iff $\forall x \in X$, $\psi_r(x)$ contains only read accesses to x and $\psi_w(x)$ contains only write accesses to x. We say that such an automaton *can read* x iff $\psi_r(x)$ is nonempty, and *can write* x iff $\psi_w(x)$ is nonempty. If Q is a collection of single-variable read-write automata, then a component of Q is said to *own* a variable x if it is the only component that can write x; in this case, x is said to be a *single-writer* variable. *Multi-writer*, *single-reader*, and *multi-reader* variables are defined in the obvious way.

Other classes of shared memory automata could be constructed in a similar manner. For example, one might define test-and-set or memory-to-memory-swap accesses and define automata in which the access labels are appropriately partitioned into additional classes. In fact, this style of definition can be used to define shared memory accesses for operations on arbitrary data types, such as enqueue and dequeue. Of course, any shared memory algorithm could be expressed and studied using the general shared memory automaton definition only, but being specific about the types of shared memory accesses allowed makes the assumptions about

the underlying shared memory more explicit, and also may help simplify reasoning about the algorithm.

7.1.4 Augmentation and Augmented-Composition

In building up I/O automaton systems, we may wish to compose collections of shared memory automata having different (either intersecting or disjoint) sets of shared variables. We would like the result of this composition to be a shared memory automaton for Z, where Z is the union of the sets of shared variables of the automata being composed. In order to accomplish this, we first "augment" each of the automata with additional shared variables so that its set of shared variables is Z. Then we compose as usual.[4]

We now define what is meant by augmenting an automaton. Let X and Z be sets of variables, with $X \subseteq Z$. Given a shared memory automaton A for X, we define $augment(A, Z)$, read "the augmentation of A to Z," to be the automaton B as follows:

- $in(B) = \{\pi \in sm\text{-}acts(Z) : \pi|X \in shared\text{-}in(A)\} \cup (in(A) \backslash shared\text{-}in(A))$.
- $out(B) = \{\pi \in sm\text{-}acts(Z) : \pi|X \in shared\text{-}out(A)\} \cup (out(A) \backslash shared\text{-}out(A))$.
- $int(B) = int(A)$.
- $states(B) = states(A)$.
- $start(B) = start(A)$.
- $steps(B)$ = all steps $\sigma = (s', \pi, s)$ such that either
  1. $\sigma \in steps(A)$ and π is not a shared memory action, or
  2. $\sigma|X \in steps(A)$ and $\pi \in shared\text{-}in(B)$, or
  3. $\sigma|X \in steps(A)$, $\pi = (f'_Z, a, f_Z) \in shared\text{-}out(B)$, and $f'_Z|(Z - X) = f_Z|(Z - X)$.
- $part(B) = \{C \subseteq local(B) : C|X \in part(A)\}$ such that $part(B)$ forms a partition of the locally-controlled actions of B.

[4] When composing a shared memory automaton with an "ordinary" I/O automaton, no augmentation is necessary, since an ordinary I/O automaton is by definition an SMA for any set of variables X.

Essentially, we augment A by making the signature complete for Z, while leaving the set of states unchanged. For each step involving a shared memory action π for X, we substitute the set of all steps in which π is replaced by a shared memory action for Z (call it π') such that $\pi'|X = \pi$. For output actions steps, we make the further restriction that if $\pi' = (f'_Z, a, f_Z)$, then f'_Z and f_Z differ only in their assignments to the variables of X. This models the fact that outputs of B only change the values of shared variables in X. We do not make this restriction for input actions because they are always enabled. This also highlights the fact that the shared memory accesses of B are independent of all shared variables other than those in X. The partition of B is constructed from that of A to reflect the differences in their signatures.

Theorem 7.2: Let X and Z be sets of variables, with $X \subseteq Z$, and let A be a shared memory automaton for X. Then $augment(A, Z)$ is a shared memory automaton for Z.

Proof: Immediate from the definitions of augmentation and shared memory automata. ■

Our next result, Theorem 7.5, says that augmentation does not (in any significant way) affect the behavior of an automaton.

Lemma 7.3: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for X and α_A is an execution of A, then there exists an execution α_B of $B = augment(A, Z)$ such that $\alpha_B|X = \alpha_A$.

Proof: Clearly, if α_A contains no actions, the claim holds. For the inductive hypothesis, let $\alpha_A = \alpha'_A \pi_A s$ be an execution of A, and let α'_B be the execution of B such that $\alpha'_B|X = \alpha'_A$. Clearly the state of A after α'_A is the same as the state of B after α'_B. Let this state be s'. It remains to be shown that some π_B is enabled from s' in B, resulting in state s, where $\pi_B|X = \pi_A$. If π_A is not a shared memory action, then the result is trivial, since the steps of A and B differ only with respect to shared memory actions. If π_A is a shared memory action (f'_X, a, f_X), then by the definition of augmentation, there must be a step $(s', \pi_B = (f'_Z, a, f_Z), s) \in steps(B)$ such that $\pi_B|X = \pi_A$. ■

Lemma 7.4: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for X and α_B is an execution of $B = augment(A, Z)$, then there exists an execution α_A of A such that $\alpha_A = \alpha_B|X$.

Proof: If α_B has no actions, the claim holds. For the inductive hypothesis, let $\alpha_B = \alpha'_B \pi_B s$ be an execution of B, and let α'_A be the execution of A such that $\alpha'_A|X = \alpha'_A$. Clearly the state of B after α'_B is the same as the state of A after α'_A. Let this state be s'. It remains to be shown that some π_A is enabled from s' in A, resulting in state s, where $\pi_A = \pi_B|X$. If π_B is not a shared memory action, then the result is trivial as before. If π_B is a shared memory action (f'_Z, a, f_Z), then by the definition of augmentation, the step $(s', (f'_Z|X, a, f_Z|X), s) \in \mathit{steps}(A)$. Therefore, the second claim holds. ∎

Theorem 7.5: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for X, then

1. $\mathit{behs}(\mathit{augment}(A, Z))|X = \mathit{behs}(A)$, and
2. $\mathit{fairbehs}(\mathit{augment}(A, Z))|X = \mathit{fairbehs}(A)$.

Proof: Part 1 is immediate from Lemmas 7.3 and 7.4.

For Part 2, let α_A be a fair execution of A, and let $\beta_A = \mathit{beh}(\alpha_A)$. From Lemma 7.3, we know that there exists an execution α_B of $B = \mathit{augment}(A, Z)$ such that $\alpha_B|X = \alpha_A$. To show that α_B is fair, we apply the definition of augmentation. From the construction of $\mathit{steps}(B)$, a shared memory action $\pi \in \mathit{acts}(B)$ is enabled in state s of B only if $\pi|X$ is enabled in state s of A. The remaining actions $\pi \in \mathit{acts}(B)$ are enabled in in state s of B only if π is enabled in state s of A. Furthermore, any two actions π and π' are in the same equivalence class of B iff $\pi|X$ and $\pi'|X$ are in the same equivalence class of A. So, since α_A is fair, α_B is fair.

Now, to show the other direction, let α_B be a fair execution of B. By Lemma 7.4, there exists an execution α_A of A such that $\alpha_A = \alpha_B|X$. To show that α_A is fair, we argue similarly to above. ∎

We can now define augmented–composition, making use of the augmentation definition and standard I/O automaton composition.

Augmented–Composition: Let $\{X_i\}_{i \in I}$ be a collection of (not necessarily disjoint) sets of variables, let $Z = \cup_{i \in I} X_i$, let each A_i be a shared memory automaton for X_i, and let the collection $\{\mathit{augment}(A_i)\}_{i \in I}$ be strongly compatible. We define the *augmented composition* $\prod^{+}_{i \in I} A_i$ to be the ordinary I/O automaton composition $\prod_{i \in I} \mathit{augment}(A_i, Z)$.

Theorem 7.6: Let $\{X_i\}_{i \in I}$ be a collection of (not necessarily disjoint) sets of variables, let $Z = \cup_{i \in I} X_i$, let each A_i be a shared memory automaton for X_i, and suppose that the collection of automata $\{augment(A_i, Z)\}_{i \in I}$ is strongly compatible. Then the augmented composition $\prod^{+}_{i \in I} A_i$ is a shared memory automaton for Z.

Proof: By Theorem 7.2, for each A_i, $augment(A_i, Z)$ is a shared memory automaton for Z. Therefore, by Theorem 7.1, the result holds. ∎

The following three compositionality results follow immediately from the corresponding results in [48], together with Theorems 7.5 and 7.6. The first result says that an execution of an augmented-composition induces executions of the component shared memory automata.

Corollary 7.7: Let $\{X_i\}_{i \in I}$ be a collection of sets of variables, where $Z = \cup_{i \in I} X_i$. Let $\{A_i\}_{i \in I}$ be a collection of automata such that each A_i is a shared memory automaton for X_i. Let the collection of automata $\{augment(A_i, Z)\}_{i \in I}$ be strongly compatible, and let $A = \prod^{+}_{i \in I} A_i$. If $\alpha \in execs(A)$ then $(\alpha | augment(A_i, Z)) | X_i \in execs(A_i)$ for every $i \in I$. Moreover, the same result holds if $execs()$ is replaced by $fairexecs()$, $scheds()$, $fairscheds()$, $behs()$, or $fairbehs()$.

The next result says that executions of component shared memory automata can often be pasted together to form an execution of the augmented-composition.

Corollary 7.8: Let $\{X_i\}_{i \in I}$ be a collection of sets of variables, where $Z = \cup_{i \in I} X_i$. Let $\{A_i\}_{i \in I}$ be a collection of automata such that each A_i is a shared memory automaton for X_i. Let the collection of automata $\{augment(A_i, Z)\}_{i \in I}$ be strongly compatible, and let $A = \prod^{+}_{i \in I} A_i$. Suppose α_i is a (fair) execution of A_i for every $i \in I$, and let β be a sequence of actions in $acts(A)$ such that $(\beta | augment(A_i, Z)) | X_i = sched(\alpha_i)$ for every $i \in I$. Then there is a (fair) execution α of A such that $\beta = sched(\alpha)$ and $\alpha_i = (\alpha | augment(A_i, Z)) | X_i$ for every $i \in I$. Moreover, the same result holds when $acts()$ and $scheds()$ are replaced by $ext()$ and $beh()$.

Finally, schedules and behaviors of component shared memory automata can also be pasted together to form schedules and behaviors of the augmented-composition.

Corollary 7.9: Let $\{X_i\}_{i \in I}$ be a collection of sets of variables, where $Z = \cup_{i \in I} X_i$. Let $\{A_i\}_{i \in I}$ be a collection of automata such that each A_i is a shared memory automaton for X_i. Let the collection of automata $\{augment(A_i, Z)\}_{i \in I}$ be strongly compatible, and let $A = \prod^{+}_{i \in I} A_i$. Let

β be a sequence of actions in $acts(A)$. If $(\beta|augment(A_i, Z))|X_i \in scheds(A_i)$ for every $i \in I$, then $\beta \in scheds(A)$. Moreover, the same result holds when $acts()$ and $scheds()$ are replaced by $ext()$ and $behs()$, respectively, and similarly when replaced by $acts()$ and $fairscheds()$ or by $ext()$ and $fairbehs()$.

7.1.5 The Closeout Operator

So far, we have introduced shared memory actions to model accesses to shared variables, and we have defined a special kind of I/O automaton containing shared memory actions in its signature. We have interpreted the first triple of each action as the "before state" of shared memory and the third component as the "after state." However, we have not yet placed any restrictions on the the relationship between the "after state" of one shared memory action and the "before state" of the next. A shared memory automaton is *not* guaranteed that the value it writes to a given shared variable will be the value observed by the next system component reading that variable. In other words, we permit the environment to freely modify the values in shared memory. We would like to construct systems in which the set of components that may modify a particular shared variable is fixed, closed to the environment. We therefore define a "closeout" operator, which takes a shared memory automaton A and produces a new automaton B such that some or all of the shared variables of A become part of the local state of B. In this way, the "absorbed" variables can be touched only the by the actions of B. Since A may be the result of composing several shared memory automata, the closeout operator achieves the desired result of restricting shared variable accesses to a particular collection of system modules.

We now define the closeout operator C. Since the state of an automaton may be thought of as a mapping from a set of variables to a set of values, we will feel free to operate on states as if they were assignment mappings. Let X and Y be disjoint sets of variables, let $Z = X \cup Y$, and let A be a shared memory automaton for Z. We define $B = C(A, X)$ as follows:

- $sig(B) = sig(A)|Y$
- $states(B) = states(A) \oplus F_X$,
- $start(B) = start(A) \oplus init(X)$,
- $steps(B)$ contains exactly the following set of steps: for each step (s', π, s) in $steps(A)$,

1. if $\pi = (f'_Z, a, f_Z)$ is a shared memory action, then
   $(s' \oplus (f'_Z|X), (f'_Z|Y, a, f_Z|Y), s \oplus (f_Z|X)) \in steps(B)$,
2. if π is not a shared memory action, then
   $\{(s' \oplus f_X,\ a,\ s \oplus f_X) : f_X \in F_X\} \subseteq steps(B)$, and

- $part(B) = part(A)$, where each class is projected on Y.

Essentially, the variables in X are absorbed into the internal state of the "closed out" automaton. If $x \in X$, we use the familiar record notation $s.x$ to refer to the value of x in a particular state s of B. That is, if $s_B = s_A \oplus f_X$, where s_A is a state of A, then $s_B.x = f_X(x)$.

Given the definition of the closeout operator, we get the following natural result.

Theorem 7.10: Let A be a shared memory automaton for Z and let X and Y be disjoint sets of variables such that $Z = X \cup Y$. Then $B = C(A, X)$ is a shared memory automaton for Y.

Proof: To show that B is an I/O automaton, we must demonstrate that for all states s' and input actions π of B, there exists a state s' of B such that $(s', \pi, s) \in steps(B)$. Since this property is true of A, and since *shared-in*(A) is complete, this property is also true of B by the construction of $steps(B)$. (When we construct the steps of B, completeness of *shared-in*(A) guarantees that we include all possible values for X in the "before states" of the steps for each input action.)

We now show that I/O automaton B is a shared memory automaton for Y. Clearly, all the shared memory actions of B are external shared memory actions for Y. We now show that each of the three conditions in the definition of a shared memory automaton hold for B. For the first condition, since *shared-in*(A) is complete for Z, *shared-in*(B) = *shared-in*$(A)|Y$ must be complete for Y. Similarly, for *shared-out*(B). The second condition requires that for every step $(s', (f'_Y, a, f_Y), s)$ in $steps(B)$, if $(f'_Y, a, f_Y) \in$ *shared-out*(B), then for all $\hat{f}'_Y \in F_Y$, there exists a state $\hat{s}$ and some $\hat{f}_Y \in F_Y$ such that $(s', (\hat{f}'_Y, a, \hat{f}_Y), \hat{s})$ is in $steps(B)$. Since this condition is true for A, we know that for each shared memory output action label a, there exists a step $(s',\ (f'_X \oplus f'_Y, a, f_X \oplus f_Y), s)$ for every possible assignment mapping $f'_X \oplus f'_Y$ for Z. Therefore, when we project on Y in constructing $steps(B)$, we have a step $(s',\ (f'_Y, a, f_Y), s)$ for each possible assignment mapping f'_Y for Y. The third condition, regarding membership of equivalence classes, is obviously true of B. ■

7.1.6 Closeout for behaviors

We now give a closeout definition for behaviors that is analogous to the one for automata.

Let X and Z be sets of variables with $X \subseteq Z$. If β is a sequence of actions of a shared memory automaton A for Z, then we say that β is *consistent for* X iff the following conditions hold:

1. if (f'_Z, a, f_Z) is the first shared memory action in β, then $f'_Z|X = init(X)$, and
2. if (f'''_Z, a_1, f''_Z) and (f'_Z, a_2, f_Z) are shared memory actions in β such that no shared memory action occurs between them, then $f''_Z|X = f'_Z|X$.

If Σ is a set of sequences of actions of a shared memory automaton for Z, then we define $C(\Sigma, X)$ to be the set $\Sigma_X|(Z - X)$, where Σ_X is the subset of Σ containing exactly those sequences that are consistent for X.

Lemma 7.11: Let X and Z be sets of variables such that $X \subseteq Z$. Let A be a shared memory automaton for Z, and let α_B be an execution of $B = C(A, X)$ with behavior β_B. Then there exists an execution α_A of A, with behavior β_A consistent for X, such that $\beta_A|(Z - X) = \beta_B$.

Proof: Let $Y = Z - X$. We construct the sequence α_A from α_B as follows. For each step $(s' \oplus f'_X, \pi, s \oplus f_X)$ in α_B, if $\pi = (f'_Y, a, f_Y)$ is a shared memory action of B, then we let the corresponding step in α_A be $(s', (f'_Y \oplus f'_X, a, f_Y \oplus f_X), s)$; and if π is not a shared memory action, we let the corresponding step in α_A be (s', π, s).

Let $\beta_B = beh(\alpha_A)$. Clearly, $\beta_B|Y = \beta_A$. It remains to be shown that α_A is an execution of A and that β_A is consistent for X. We show that α_A is an execution of A by showing that each step of α_A is in $steps(A)$. Let $\sigma = (s' \oplus f'_X, \pi, s \oplus f_X)$ be a step of B. If $\pi = (f'_Y, a, f_Y)$ is a shared memory action of B, then by the construction of $steps(B)$ in the definition of closeout, $(s', (f'_Y \oplus f'_X, a, f_Y \oplus f_X), s)$ must be a step of A. Similarly, if π is not a shared memory action, then (s', π, s) must be a step of A. Therefore, the construction produces an execution of A.

Finally, we show that β_A is consistent for X. Since every initial state of $C(A, X)$ is in $states(A) \oplus init(X)$, it must be that the first shared memory action (f'_Z, a, f_Z) of β_B has $f'|X = init(X)$, so the first consistency condition is satisfied. We know that the second consistency condition must be satisfied, since any two successive steps (s''', π_1, s'') and (s', π_2, s)

of any execution must have $s'' = s'$, the assignments to the variables of X are part of the state of $C(A, X)$, and the only actions that may change the values for X in the state of $C(A, X)$ correspond to shared memory actions for for Z. ■

Lemma 7.12: Let X and Z be sets of variables such that $X \subseteq Z$. Let A be a shared memory automaton for Z and let α_A be an execution of A with behavior β_A. If β_A is consistent for X, then there exists an execution α_B of $B = C(A, X)$ such that $\beta_A|(Z - X)$ is the behavior of α_B.

Proof: Let $Y = Z - X$. Let α_B be the execution constructed from α_A as follows. For each shared memory action π in α_A, let the corresponding action in α_B be $\pi|Y$. Leave the remaining actions as in α_A. For each state s in α_A, let the corresponding state in α_B be $s \oplus (f_Z|X)$, where f_Z is the third component of the preceding shared memory action in α_A (or $f_Z = init(Z)$ if there is no preceding shared memory action).

Clearly $\beta_A|Y = beh(\alpha_B)$. We claim that α_B is an execution of B. To prove this claim, we proceed by induction on the length of α_B, showing that each action is enabled from the state in which it occurs. Clearly, if α_B contains no actions, then the claim holds. Let (s'_A, π, s_A) be a step of α_A, and let α'_B be the portion of α_B up to (but not including) the action $\pi|Y$ for the corresponding step in α_B. We wish to show that if α'_B ends in state s'_B, then the step $(s'_B, \pi|Y, s_B) \in steps(B)$, where s_B is the next state of α_B. By the construction, we know that $s'_B = s'_A \oplus (f'_Z|X)$, where f'_Z is the third component of the preceding shared memory action in α_A (or $f'_Z = init(Z)$ if there is no preceding shared memory action), and similarly for s_B. There are two cases for π:

1. If π is not a shared memory action, then clearly it is enabled from s'_B, since (by the construction) s'_A and s'_B are identical except that s'_A does not assign values to the variables in X. Furthermore, since π is not a shared memory action, $s_B|X = s'_B|X$, so the step exists by the definition of the closeout operator.

2. If $\pi = (f'_Z, a, f_Z)$ is a shared memory action, then consistency of β_A requires that f'_Z be the third component of the preceding shared memory action in α_A (or $init(Z)$ if there is no such preceding action). By the definition of closeout, we know $steps(B)$ contains the step $(s'_A \oplus (f'_Z|X), (f'_Z|Y, a, f_Z|Y), s_A \oplus (f_Z|X))$. And by the construction, $s'_A \oplus (f'_Z|X) = s'_B$ and $s_A \oplus (f_Z|X) = s_B$. Therefore, the desired step exists.

In both cases, $\pi|Y$ is enabled and leads to state s_B. ■

Theorem 7.13: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for Z, then

1. $behs(C(A,X)) = C(behs(A),X)$, and
2. $fairbehs(C(A,X)) = C(fairbehs(A),X)$.

Proof: Part 1: Let $Y = Z - X$. By Lemma 7.11, we know that if $\beta|Y$ is a behavior of $C(A,X)$, then β is a behavior of A that is consistent for X. Therefore $\beta|Y \in C(behs(A),X)$, by definition. If $\beta|Y \in C(behs(A),X)$, then by definition of closeout on behaviors, β is consistent for X. Therefore, Lemma 7.12 tells us that $\beta|Y \in behs(C(A,X))$.

Part 2: First, we show that $fairbehs(C(A,X))$ contains $C(fairbehs(A),X)$. Let β_B be a fair behavior of $B = C(A,X)$, and let α_B be an execution of B with $beh(\alpha_B) = \beta_B$. Construct execution α_A of A from α_B as in the proof of Lemma 7.11 such that $beh(\alpha_A)|(Z - X) = \beta_B$. Since A is a shared memory automaton, we know that *shared-out*(A) is complete and that for any given access label $a \in \mathcal{L}$, all shared memory actions with label a belong to the same class. Furthermore, by the definition of closeout, π_A and π'_A belong to the same equivalence class in A iff $\pi_A|X$ and $\pi'_A|X$ belong to the same equivalence class in B. Therefore, given that α_B is fair, we can show that α_A is fair by arguing that an action π_A is enabled in state s_A of α_A iff $\pi_A|X$ is enabled in the corresponding state s_B of α_B. This is easily seen from the construction of $steps(B)$, since $s_A = s_B|(Z - X)$.

Now, we show that $C(fairbehs(A),X)$ contains the set $fairbehs(C(A,X))$. Let β_A be a fair behavior of A that is consistent for X, and let α_A be an execution of A with $beh(\alpha_A) = \beta_A$. Construct execution α_B of $C(A,X)$ from α_A as in the proof of Lemma 7.12 such that $\beta_A|(Z - X) = beh(\alpha_B)$. The remainder of the proof is argued as above. ■

7.1.7 Discussion

Important in defining our shared memory extensions were the built-in features of the I/O automaton model, most notably composition and the separation of inputs and outputs. By using the built-in notion of an output action being under the control of a single process, we were able to capture the idea of a single module making an atomic update to shared memory

(without any active participation by other modules). In addition, by exposing the values of the shared variables as part of the shared memory accesses, we were able to not only carry forward the compositionality properties of I/O automaton behaviors but also provide a useful notion of a shared memory action as an input. We expect normal communication through shared variables to be modeled using output actions only, but the input actions allow a module to passively observe the accesses to shared memory made by other processes. We see two potential uses for this feature. First, one might use shared memory actions as inputs to construct external processes that are not part of the algorithm but monitor the use of shared memory (possibly as a means to check algorithms in a simulation system). Second, in a modular algorithm design, it may be appropriate to divide a task into several I/O automaton components such that only one component accesses the shared memory while the others are kept "informed" of these accesses by receiving them as inputs (e.g., to model a collection of processes "snooping" on a memory bus to update local caches).

7.2 Example: Dijkstra's mutual exclusion algorithm

In order to illustrate the shared memory extensions just presented, we apply them to Dijkstra's classical shared memory mutual exclusion algorithm. We begin by defining the mutual exclusion problem in terms of the I/O automaton model. We then present Dijkstra's algorithm as a composition of shared memory automata. The safety and progress proofs that follow demonstrate how proofs using standard assertional techniques may be expressed straightforwardly using this model.

7.2.1 The Mutual Exclusion Problem

Fix n, a positive integer, and let $\mathcal{I} = \{1, 2, \ldots, n\}$. We define schedule module M with $sig(\mathrm{M})$ as follows:

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

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 $\mathrm{UserTry}_i$ and $\mathrm{UserExit}_i$, 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 and user-live sequences of actions of M. Let β be a sequence of actions in $sig(\text{M})$. 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. If β is a sequence of actions of S, we say that β is *user-live* iff for all $i \in \mathcal{I}$, $\beta|i$ is either infinite or does not end with Crit_i. Informally, this says that no user u_i stops in its critical section. An execution is said to be well-formed (user-live) iff its behavior is well-formed (user-live).

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

1. M preserves well-formedness in β.
2. If β is well-formed, then
   (a) (mutual exclusion) $\forall i, j \in \mathcal{I}$, if Crit_i and Crit_j occur in β (in that order), then UserExit_i occurs between them.
   (b) (progress) if β is user-live, then either β is infinite or $\forall i, \beta|i$ ends with Rem_i.

Condition (2a) is the safety property: no two processes are in their critical sections simultaneously. Condition (2b) is the progress property: either all processes eventually end up in their remainder regions or some process enters the critical region infinitely often. Both properties are guaranteed only if the user processes preserve well-formedness, and the progress condition is guaranteed only if user processes eventually exit the critical region. In this variant of the mutual exclusion problem, only a very weak progress requirement is made. For example, correct solutions to this problem admit executions in which a process is locked out of the critical section.

7.2.2 Dijkstra's Mutual Exclusion Algorithm

In this section, we model Dijkstra's shared memory mutual exclusion algorithm [15] as an illustration of our shared memory extensions to the I/O automaton model. As presented here, the variable names and structure more closely follow the description in [45], although the algorithm is the same.

We implement schedule module M by a collection of n automata p_i, $i \in \mathcal{I}$, where each p_i interacts with u_i through shared actions and interacts with the other p_j's using shared variables. Each p_i has three state components: *stage* $\in$ {try, read, check, set, control2, final_check, failed, crit, exit, done, remainder}; k, an integer in the range 1 to n; and *checked*, a set of integers in the range 1 to n. Initially, *stage* = remainder, k is arbitrary, and *checked* is the empty set. Automaton p_i is a shared memory automaton for V, where V has the following variables: k, an integer in the range 1 to n; and *control*$[j]$ for $j \in \mathcal{I}$, which take on values from {0,1,2}. Initially, k has an arbitrary value and all *control* variables are 0. The code for automaton p_i is shown in Figure 7-1. Shared memory actions are listed by their access labels and distinguished by daggers (†); all other actions are listed by their action names. All actions of p_i are outputs, except UserTry$_i$ and UserExit$_i$, which are its inputs. "Pre" and "Eff" denote "precondition" and "effect", respectively. For shared memory actions, the step $(s', (v', a, v), s)$ is in *steps*(p_i) exactly when the precondition for a is satisfied in state s' and s and v are derived from s' and v' according to the effect clause. For all other output actions, the step (s', π, s) is in *steps*(p_i) exactly when the precondition for π is satisfied in state s' and state s is derived from state s' according to the effect clause. If an action has no precondition, it is always enabled. If a state component or variable is not mentioned in the effect clause, it is left unchanged by the action. The partition consists of a class for each $i \in \mathcal{I}$ that contains all the output actions of p_i.

Essentially, the algorithm proceeds in two stages. After receiving a UserTry$_i$ input, p_i sets its control variable to 1 and enters stage one. In stage one, it continually reads k and checks to see if control[k] is 0. If it finds a 0 in control[k], it sets k to its own index i. If it reads k and finds it equal to i, p_i proceeds to stage two and sets its control variable to 2. In stage two, p_i performs a final check by examining the control variables of all the other processes. If any of these control variables are found to be 2, then p_i fails and returns to stage one (where it sets its control variable back 1). Otherwise, p_i finds all the control variables to be less than 2 and

- UserTry$_i$
  Eff: $s.\text{stage}_i = \text{try}$

† Try$_i$
  Pre: $s'.\text{stage}_i \in \{\text{try, failed}\}$
  Eff: $v.\text{control}[i] = 1$
  $s.\text{checked}_i = \{i\}$
  $s.\text{stage}_i = \text{read}$

† Read$_i$
  Pre: $s'.\text{stage}_i = \text{read}$
  Eff: $s.k_i = v'.k$
  if $s.k_i = i$ then
  $s.\text{stage}_i = \text{control2}$
  else
  $s.\text{stage}_i = \text{check}$

† Check$(j)_i$
  Pre: $s'.\text{stage}_i = \text{check}$
  $j = s'.k_i$
  Eff: if $v'.\text{control}[j] = 0$ then
  $s.\text{stage}_i = \text{set}$
  else
  $s.\text{stage}_i = \text{read}$

† Set$_i$
  Pre: $s'.\text{stage}_i = \text{set}$
  Eff: $v.k = i$
  $s.\text{stage}_i = \text{read}$

† Control2$_i$
  Pre: $s'.\text{stage}_i = \text{control2}$
  Eff: $v.\text{control}[i] = 2$
  $s.\text{stage}_i = \text{final\_check}$

† FinalCheck$(j)_i$
  Pre: $s'.\text{stage}_i = \text{final\_check}$
  $j \notin s'.\text{checked}_i$
  Eff: if $v'.\text{control}[j] = 2$ then
  $s.\text{stage}_i = \text{failed}$
  else
  $s.\text{checked}_i = s'.\text{checked}_i \cup \{j\}$

- Crit$_i$
  Pre: $s'.\text{stage}_i = \text{final\_check}$
  $|s'.\text{checked}_i| = n$
  Eff: $s.\text{stage}_i = \text{crit}$

- UserExit$_i$
  Eff: $s.\text{stage}_i = \text{exit}$
  $s.\text{checked}_i = \{i\}$

† Reset$_i$
  Pre: $s'.\text{stage}_i = \text{exit}$
  Eff: $v.\text{control[i]} = 0$
  $s.\text{stage}_i = \text{done}$

- Rem$_i$
  Pre: $s'.\text{stage}_i = \text{done}$
  Eff: $s.\text{stage}_i = \text{remainder}$

Figure 7-1: Transition Relation for p_i in Dijkstra's Algorithm

issues a Crit_i action, allowing u_i to proceed to the critical section. After u_i leaves the critical section (and issues a UserExit_i action), p_i resets its control variable to 0 and issues a Rem_i action.

We associate with each p_i an access partition ψ^i as follows: For each $j \in \mathcal{I}$, $\psi^i_r(control[j])$ contains the labels $\text{Check}(j)_i$ and $\text{FinalCheck}(j)_i$. Also, $\psi^i_w(control[i])$ contains the labels Try_i, Control2_i, and Reset_i. And for each $j \neq i$, $\psi^i_w(control[j])$ is empty. Finally, $\psi^i_r(k)$ contains Read_i and $\psi^i_w(k)$ contains Set_i. The following result follows immediately from inspection of the code.

Lemma 7.14: For all $i \in \mathcal{I}$, automaton p_i is a single-variable read-write automaton under ψ^i.

We let system $S = C(\Pi_{1 \leq i \leq n} p_i, \{k, control[i], i \in \mathcal{I}\})$ be the composition of the processes of Dijkstra's algorithm, closed out on k and the *control* variables. Furthermore, we hide all shared memory actions of S so that the external signatures of M and S are the same. One may note that all the p_i's in system S can read and write shared variable k, whereas the variable control[i] may be written only by p_i and read by the other p_j's. That is, each *control*[i] is owned by p_i, while k is a multi-writer variable.

We wish to show that system S solves schedule module M. The proof has three parts. First, we show that S preserves well-formedness in all executions, Condition (1) of module M. In Section 7.2.3, we give the safety proof, Condition (2a). Finally, we present the progress proof, Condition (2b), in Section 7.2.4.

If i is a process index and s is a state of system S, we say that process p_i is a *contender* in state s, written *contender*(i, s), iff $s.\text{stage}_i \in$ {read, check, set, control2, final_check, failed}.

Lemma 7.15: Let α be an execution of system S with behavior β. Then system S preserves well-formedness in β.

Proof: By induction on the length of α. For the base case, if α contains no actions, then it is well-formed. Let $\alpha = \alpha' s \pi$, where $beh(\alpha')$ is well-formed and π is an output action of S. There are two cases.

- If π is a Crit_i action, then by the preconditions of that action it must be that p_i is a contender in state s. Therefore, the last action in $beh(\alpha')|i$ must be UserTry_i, for any other action would leave p_i in a non-contender state.

- If π is a Rem_i action, then by the preconditions of that action it must be that stage_i = done in state s. Therefore, the last action in $\beta'|i$ must be Reset_i, for any other action would leave p_i in a state with $\text{stage}_i \neq$ done. Since Reset_i is only enabled when stage_i = exit, the last action in $beh(\beta')|i$ must be UserExit_i, for any other action would leave p_i in a state with $\text{stage}_i \neq$ exit.

In both cases, β is well-formed. ■

The following lemma will be used in the safety proof to rule out the occurrence of UserTry and UserExit actions from certain states.

Lemma 7.16: Let α be an execution of system S with behavior β. If β is well-formed, then for all states s in α, if s is immediately followed by a UserTry_i (UserExit_i) action, then $s.\text{stage}_i$ is remainder (crit).

Proof: If s is followed by UserTry_i, then by definition of well-formedness, the preceding action in $\beta|i$ is a Rem_i action, and a Rem_i action leaves stage_i = remainder. Furthermore, no output actions of p_i are enabled while stage_i = remainder. If s is followed by UserExit_i, then by definition of well-formedness, the preceding action in $\beta|i$ is a Crit_i action, and a Crit_i action leaves stage_i = crit. Furthermore, no output actions of p_i are enabled while stage_i = crit. ■

7.2.3 Safety Proof

Let s be a state of system S. To denote the set of processes in (or about to enter) their critical sections, we define the set $ready(s) = \{i : (s.\text{stage}_i = \text{crit}) \vee (|s.\text{checked}_i| = n)\}$. The proof is based on a set of invariants, proved in the following Lemma.[5]

Lemma 7.17: Let α be a well-formed execution of system S. In states s of α, for all processes p_i and p_j, the following facts hold:

1. $s.\text{control}[i] = 2$ iff $s.\text{stage}_i \in \{\text{final\_check, failed, crit, exit}\}$.

2. If $s.\text{checked}_i \neq \{i\}$ then $s.\text{stage}_i \in \{\text{final\_check, failed, crit}\}$.

[5] Although the last invariant of Lemma 7.17 is used only in the liveness proof, we present it here because of its similarity to the others.

3. If $i \neq j$, then $i \in s.\text{checked}_j \Rightarrow j \notin s.\text{checked}_i$.

4. If $i \in \text{ready}(s)$ then $s.\text{checked}_i = \{1, 2, ...n\}$.

5. If $s.\text{stage}_i \in$ {control2, final_check, failed, crit, exit, done}, then $s.k_i = i$.

Proof: In the initial state of S, $\forall i \in \mathcal{I}$, control[$i$] = 0, checked$_i = \{i\}$, and stage$_i$ = remainder. Therefore, all the facts hold in the initial state. Let $\alpha = \alpha'\pi s$, and assume that the facts hold in all states of α', and specifically in the last state s' of α'. We consider each fact in turn, showing that it must hold in state s as well.

1: If $s'.\text{control}[i] = 2$, then by the induction hypothesis $s'.\text{stage}_i \in S =$ {final_check, failed, crit, exit}. Therefore, π must be either Try$_i$, FinalCheck$(j)_i$, Crit$_i$, UserExit$_i$, or Reset$_i$. (Lemma 7.16 rules out UserTry$_i$.) The actions FinalCheck, Crit$_i$, and UserExit$_i$ do not change the value of control[i] and result in $s.\text{stage}_i \in S$. The actions Try$_i$ and Reset$_i$ both cause $s.\text{control}[i] \neq 2$, but also result in $s.\text{stage}_i \notin S$. Therefore, the property is preserved if $s'.\text{control}[i] = 2$.

If $s'.\text{control}[i] \neq 2$, then by the induction hypothesis $s'.\text{stage}_i \notin S$. Therefore, π must be either UserTry$_i$, Try$_i$, Read$_i$, Check$(j)_i$, Set$_i$, Control2$_i$, or Rem$_i$. (By Lemma 7.16, UserExit$_i$ is ruled out.) Actions UserTry$_i$, Read$_i$, Check$(j)_i$, Set$_i$, and Rem$_i$ do not change the value of control[i] and result in $s.\text{stage}_i \notin S$. Furthermore, the action Try$_i$ sets control[i] = 1 and results in $s.\text{stage}_i \notin S$. Finally, the action Control2$_i$ sets control[i] = 2, but also results in $s.\text{stage}_i \in S$. Therefore, the property is preserved if $s'.\text{control}[i] \neq 2$.

2: If $s'.\text{checked}_i = \{i\}$, then the only possibility for π which could cause $s'.\text{checked}_i \neq \{i\}$ is FinalCheck$(j)_i$. This action is only enabled if $s'.\text{stage}[i] =$ final_check. The FinalCheck$(j)_i$ either does not change stage$_i$ or sets $s.\text{stage}_i =$ failed. Therefore, the property is preserved.

If $s'.\text{checked}_i \neq \{i\}$, then by the induction hypothesis, $s'.\text{stage}_i \in$ {final_check, failed, crit}. Therefore, the only possibilities for π which could cause $s.\text{stage}_i \notin$ {final_check, failed, crit} are Try$_i$ and UserExit$_i$. (The action UserTry$_i$ is ruled out by Lemma 7.16.) However, in both cases, $s.\text{checked}_i = \{i\}$, so the property is preserved.

3: The proof is by contradiction. Suppose $\exists i \neq j$ such that $i \in s.\text{checked}_j$ and $j \in s.\text{checked}_i$. Without loss of generality, suppose that $i \in s'.\text{checked}_j$, and let π be the action that adds j to checked_i. (By the induction hypothesis, we know that $j \notin s'.\text{checked}_i$.) The only possibility for π is $\text{FinalCheck}(j)_i$. By the transition relation, π can only add j to checked_i if $s'.\text{control}[j] \neq 2$. However, by the induction hypothesis (Fact 2), we know that $s'.\text{stage}_j \in \{\text{final\_check, failed, crit}\}$, since $s'.\text{checked}_j \neq \{j\}$. Therefore, by Fact 1, we know that $s'.\text{control}[j] = 2$, a contradiction.

4: Recall, from the definition, that $i \in \text{ready}(s)$ iff $s.\text{stage}_i = \text{crit} \vee |s.\text{checked}_i| = n$. By a pigeonhole argument, the fact clearly holds when $|s.\text{checked}_i| = n$. If $s'.\text{stage}_i \neq \text{crit}$, then the only possibility for π to make $s.\text{stage}_i = \text{crit}$ is the Crit_i action. That action has as a precondition that $|\text{checked}_i| = n$, and does not change the value of checked_i. Therefore, the property is preserved. If $s'.\text{stage}_i = \text{crit}$, then the only possibility for π to make $|s.\text{checked}_i| \neq n$ is UserExit_i, but this also results in $\text{stage}_i = \text{exit}$.

5: If $s'.\text{stage}_i \in \{\text{control2, final\_check, failed, crit, exit, done}\}$, then by the inductive hypothesis, $s'.k_i = i$. Furthermore, the only action which can change k_i is a Read_i action, which is only enabled if $\text{stage}_i = \text{read}$, so $s.k_i = s'.k_i = i$. If $s'.\text{stage}_i \notin \{\text{control2, final\_check, failed, crit, exit, done}\}$, then the only possible action for π which could cause $s.\text{stage}_i$ to be in that set is a Read_i action. (Lemma 7.16 rules out UserExit_i.) However, the Read_i action can only set $s.\text{stage}_i = \text{control2}$ if $s.k_i = i$. Thus, the property is preserved.

All five facts hold in state s. ■

We can now show that no two processes may be in (or about to enter) their critical sections.

Theorem 7.18: If s is a state of system S, $|\text{ready}(s)| \leq 1$.

Proof: By contradiction. Suppose that $|\text{ready}(s)| > 1$. Then by Fact 4 of Lemma 7.17, there must exist two processes p_i and p_j such that $s.\text{checked}_i = s.\text{checked}_j = \{1, 2, ...n\}$. However, this contradicts Fact 3 of Lemma 7.17. ■

It follows that the algorithm satisfies mutual exclusion.

Corollary 7.19: Let α be a well-formed execution of system S. Then $\forall i, j \in \mathcal{I}$, if Crit_i and Crit_j occur in α (in that order), then UserExit_i occurs between them.

Proof: By well-formedness and inspection of the code for system S, if a Crit_i action occurs in α then $\text{stage}_i = \text{crit}$ in all states up until the next UserExit_i action. Suppose (for contradiction) that there exist two processes p_i and p_j such that Crit_i and Crit_j occur in α (in that order) and no UserExit_i occurs between them. Then after Crit_j occurs, $\text{stage}_i = \text{crit}$ and $\text{stage}_j = \text{crit}$. However, by Theorem 7.18 and the definition of *ready*, this is impossible. ■

7.2.4 Progress Proof

In this section, we show that Dijkstra's algorithm makes progress: if a process is attempting to enter the critical section, then eventually it or some other process will enter the critical section. We define a "no progress execution" of system S and then show that no such executions exist. The proof is by contradiction: We define a well-founded variant function, or progress metric. Then we show that in no-progress executions the function is nonincreasing and must eventually decrease. Since no infinite-length decreasing chains are possible, this shows that no-progress executions do not exist. The notion of fairness, which we inherit "for free" from the original I/O automaton model, is used to show that the variant function eventually decreases.

Let $\gamma = \alpha\beta$ be a fair well-formed user-live execution of system S. Furthermore, let none of the following actions occur in β: UserTry, Crit, UserExit, Rem. If β begins with a state in which some process has stage $\neq$ remainder, then β is said to be a *no-progress execution suffix* and γ is said to be a *no-progress execution.*

Lemma 7.20: Let β be a no-progress execution suffix, and let s be a state in β. Then $\forall i \in \mathcal{I}$, $s.\text{stage}_i \notin \{\text{crit, exit, done}\}$.

Proof: Immediate from the definitions of no-progress execution suffix, fairness, and p_i. ■

Before defining the variant function, we identify an important predicate on system states. If s is a state of S, we say that s is *consistent*, denoted *consistent*(s), iff for all $i \in \mathcal{I}$, $\text{contender}(i, s) \Rightarrow s.k_i = s.k$.

We now define the variant function. Given state s of system S, we define

$$f(s) = (A, B, C, D, E, F, G, H, I, J, K),$$

where each tuple component has the nonnegative integer value defined as follows:

$A = |\{i : s.\text{stage}_i = \text{try}\}|$.

$B = |\{i : s.\text{stage}_i = \text{read}\}|$ if $\neg\text{contender}(s.k, s)$,
0 otherwise.

$C = |\{i : s.\text{stage}_i = \text{check} \wedge \neg\text{contender}(s.k_i, s)\}|$.

$D = 0$ if $\text{contender}(s.k, s)$, 1 otherwise.

$E = |\{i : s.\text{stage}_i = \text{set}\}|$.

$F = |\{i : s.\text{stage}_i = \text{control2} \wedge i \neq s.k\}|$.

$G = |\{i : s.\text{stage}_i = \text{final\_check} \wedge i \neq s.k\}|$.

$H = |\{i : \text{contender}(i, s) \wedge k_i \neq s.k\}|$.

$I = \Sigma_i(n - |\text{checked}_i|)$, for all $i \neq s.k$ such that
$s.\text{stage}_i = \text{final\_check}$.

$J = |\{i : s.\text{stage}_i = \text{failed} \wedge i \neq s.k\}|$.

$K = n$ if $(\neg\text{consistent}(s) \vee s.\text{stage}_{s.k} \neq \text{final\_check})$,
$n - |\text{checked}_{s.k}|$ otherwise.

We define a lexicographic total order on the elements in the range of f. We will show that f is nonincreasing and will eventually decrease in a no-progress execution suffix, but first we explain the components of f. The components A, E, F, G, and J simply count the number of processes in a certain stage (in some cases ignoring the process whose index is the value of the shared variable k). These components measure progress of the contenders through the different stages of the algorithm. Components B and C serve a similar purpose for the "read" and "check" stages, but are more complicated because contenders may cycle through these two stages while they wait for some other process to "get out of the way." Component B's purpose is to count the number of processes in the "read" stage; however, when the shared variable k is the index of a contender, $B = 0$. In this way, the value of B does not increase when a contender "backs off" to read k again. Component C counts the number of processes in the "check" stage whose local k variables contain indices of non-contenders.

Component D becomes 0 when the shared variable k is set to the index of a contender, and remains 1 otherwise. Components I and K both count down the number of indices that are missing from the *checked* sets of processes whose stage is "final_check." Component I hold the sum of this count for all the contenders whose indices are not equal to the shared variable k. Component K holds this count for the (at most one) contender whose index is equal to the shared variable k, but only starts counting down after all other contenders are "out of the way," meaning that their local k's are equal to the shared k.

In studying the variant function, two important progress "landmarks" should be noted. The first is when component D reaches 0, after which point the value of k is always the index of a contender. After D reaches 0, the second landmark is when component H reaches 0, meaning that all later states are consistent. After this point, all processes other than p_k cannot escape the Read-Check cycle, so nothing stands in p_k's way.

We now show that the value of the variant function f is nonincreasing in no-progress execution suffixes, and that only certain steps leave f unchanged.

Lemma 7.21: Consider any state s' in β, a no-progress execution suffix. If action π of process p_i occurs from state s' producing state s, then the following conditions hold:

1. $f(s') \geq f(s)$, and
2. either $f(s') > f(s)$ or one of the following hold:
   (a) π is a Read action and $s'.k_i = s'.k$, a contender, or
   (b) π is a Check action and $s'.k_i$ is a contender, or
   (c) π is a Try action, $i = s'.k$, and $s'.\text{stage}_i = \text{failed}$, or
   (d) π is a Control2 action and $i = s'.k$, or
   (e) π is a FinalCheck action, $i = s'.k$, and $\neg\text{consistent}(s')$.

Condition (1) says that the variant function is nonincreasing. Condition (2) says that every action must decrease the variant function, except for a few special cases. Exceptions (2a) and (2b) handle the case of a process cycling through the "read" and "check" stages, waiting for

some other process to get out of the way. Note the relationship between items (2a) and (2b) and the variant function components B and C, respectively. A process does not make progress when it reads the same value of the shared variable k that it read the previous time. Similarly, a process does not make progress if it discovers that the control variable corresponding to its local k is nonzero. Exceptions (2c), (2d), and (2e) pertain only to the contender whose index is the value of the shared variable k. Process p_k may "back off" several times before it finally enters the critical section, and the variant function is carefully constructed not to change when p_k backs off. These last three exceptions are the necessary result.

Proof: By case analysis. For each possible action, we note the changes in the components of the variant function f. (We will use A' and A to denote the first components of $f(s')$ and f(s), respectively. Similarly for B' and B, etc.) Each case may be verified by Lemma 7.20 and inspection of the preconditions and effects of the action under consideration.

- If $\pi = (v', \text{Try}_i, v)$, there are three cases:

  (1) If $s'.\text{stage}_i = \text{try}$, then $A' > A$, decreasing f.

  (2) If $s'.\text{stage}_i = \text{failed}$ and $i \neq s'.k$, then $J' > J$, and no components increase. (Component B cannot increase because Fact (5) from Lemma 7.17 tells us that if $\text{stage}_i = \text{failed}$, then $k_i = i$, a contender by definition.) Therefore, f is decreased.

  (3) If $s'.\text{stage}_i = \text{failed}$ and $i = s'.k$, then $f(s') = f(s)$, satisfying Condition 1 and exception 2c.

- If $\pi = (v', \text{Read}_i, v)$, there are three cases:

  (1) If $s'.k$ is not a contender, then $B' > B$ and A is unchanged, so f decreases.

  (2) If $s'.k_i \neq s'.k$, then $H' > H$ and no higher order components are increased, so f decreases.

  (3) If $s'.k_i = s'.k$, a contender, then $f(s') = f(s)$, satisfying Condition 1 and exception 2a.

- If $\pi = (v', \text{Check}(j)_i, v)$, there are two cases:

  (1) If $s'.k_i$ is a contender, then $f(s') = f(s)$, satisfying Condition 1 and exception 2b.

  (2) Otherwise, $C' > C$, and A and B are unchanged, so f is decreased.

- If $\pi = (v', \text{Set}, v)$, then $B = 0$, $D = 0$, $E' > E$, and A and C are unchanged. Therefore f decreases.

- If $\pi = (v', \text{Control2}_i, v)$, there are two cases:

  (1) If $i = s'.k$ then $f(s') = f(s)$, satisfying Condition 1 and exception 2d.

  (2) Otherwise, $F' > F$ and no higher order components are changed, so f decreases.

- If $\pi = (v', \text{FinalCheck}(j)_i, v)$, there are three cases:

  (1) If $i \neq s'.k$, then $I' > I$ and no higher order components are changed, so f decreases.

  (2) If $i = s'.k$ and $\neg$consistent(s'), then $f(s') = f(s)$, so Condition 1 and exception 2d are satisfied.

  (3) If $i = s'.k$ and consistent(s'), then K is the only component that may change. Suppose, for contradiction, that K does not decrease. By the effects of FinalCheck and the definition of K, the only way for this to happen is for $s'.\text{control}[j] = 2$. If $s'.\text{control}[j] = 2$, then Fact 1 of Lemma 7.17 tells us that $s'.\text{stage}[j] \in \{\text{final\_check}, \text{failed}, \text{crit}, \text{exit}\}$. Therefore, by Fact 5 of the same Lemma, $s'.k_j = j$. Since s' is consistent, $s'.k_j = s'.k$, and we have stated that $s'.k = i$. So, by transitivity, $j = i$. By the preconditions on FinalCheck, $j \notin s'.\text{checked}_i$. But $i \in s'.\text{checked}_i$, since $i \in \text{checked}_i$ initially and no action may remove it from that set. Therefore $j \neq i$, a contradiction.

In each case, the Lemma holds. The set of cases is complete by Lemma 7.20 and the definition of a no-progress execution. ■

We have just shown that the value of the variant function f never increases in a no-progress execution suffix, and that only certain steps leave its value unchanged. Now we will show that a fair execution cannot proceed using only those certain steps, so the function must eventually decrease.

Corollary 7.22: Let α be a no-progress execution suffix. Then f must eventually decrease in α.

Proof: Suppose that f is fixed in α', a suffix of α. Then, by Lemma 7.21 for all states s' of α', if π occurs from s', then one of the following hold:

- π is a Read action and $s'.k_i = s'.k$, a contender, or

- π is a Check action and $s'.k_i$ is a contender, or
- π is a Try action, $i = s'.k$, and $s'.\text{stage}_i = \text{failed}$, or
- π is a Control2 action and $i = s'.k$, and
- π is a FinalCheck action, $i = s'.k$, and $\neg\text{consistent}(s')$.

Since no action in α' is a Set action, the shared variable k is fixed in α'. *Fairness tells us that all contenders must continue taking steps.* (Inspection of the code will reveal that a contender always has *some* step enabled.) Therefore, by the four conditions above, all contenders other than p_k must have stage $\in$ {read, check}; otherwise their steps would decrease the value of f, contradicting our assumption that f is fixed. Therefore, by the same fairness argument, a Read action must eventually occur for each of these contenders, after which point its local value of k matches the shared k.

Let α'' be the suffix of α' after which all contenders other than p_k have their local k's equal to the shared k. Now, consider p_k, which must continue to take steps in α'', and let s'' be a state in α'' from which p_k takes a step. If p_k takes a FinalCheck step, then by Fact 5 of Lemma 7.17, $s''.k_k = s''.k$. However, this implies that s'' is consistent. Therefore, the conditions above imply that no FinalCheck actions can occur. If p_k takes a Control2 step, then a FinalCheck action will become enabled and remain enabled until it occurs, so fairness tells us that a FinalCheck action will eventually occur, but we have just ruled this out. The only remaining actions for p_k are Read, Check, and Try. If p_k takes a Read step, then it will observe that the shared k contains its own index and proceed to stage = control2, meaning that it must eventually take a Control2 step, which we have already ruled out. If p_k takes a Check step, then since (by statement 2 above) $s''.k_k$ is a contender, it will proceed to stage = read, meaning that it must eventually take a Read step, which we have just ruled out. Finally, if p_k takes a Try step, it will also proceed to stage = read. Therefore, if p_k continues to take steps, it eventually will decrease the value of f, giving us our contradiction. ■

Our main liveness result follows immediately.

Theorem 7.23: The set of no-progress executions for Dijkstra's algorithm is empty.

Proof: By Lemma 7.21, we know that the value of the variant function f is nonincreasing in a no-progress execution suffix. Furthermore, by Lemma 7.22, the value of f never reaches a fixed point. Therefore, since f cannot infinitely decrease, the theorem holds. ■

Finally, we show that the above theorem implies that Dijkstra's algorithm satisfies the required progress property.

Corollary 7.24: Let α be a fair well-formed user-live execution of system B. Then either $\forall i, \alpha|i$ ends with Rem_i, or $\exists i$ such that $\alpha|i$ is infinite.

Proof: By contradiction. Suppose that α is finite and that there exists some $l \in \mathcal{I}$ such that $\alpha|l$ does not end with Rem_i. Then there exists a suffix of α in which p_l has stage $\neq$ remainder and $\alpha|i$ is empty for all i. This is a no-progress execution suffix, by definition. Therefore α is a no-progress execution, which is a contradiction of Theorem 7.23. ■

7.3 Proofs for Shared Object Systems

It is convenient to use shared atomic objects as system components when building large concurrent systems. Each operation on an atomic object appears to execute indivisibly, thereby allowing the programmer to consider only interleavings of the operations, rather than their true concurrency. For performance reasons, however, it is often useful to allow concurrency between operations on a single object, so the condition of atomicity is only on an object's behavior, not on its implementation.

There are two general approaches to modelling shared objects. Which is more natural depends upon whether the intent is to model a system that *uses* atomic objects, or one that *implements* them. In a system that uses atomic objects, it is convenient to represent each operation as a single shared action between the object and the invoking process; we call these *atomic access systems.* In a system that implements atomic objects, each operation can be modeled by an invocation event and response event, denoting, respectively, the beginning and end of operation execution; we call these *invocation-response systems.*

In an invocation-response system, it is possible to consider operation executions that overlap in time. Herlihy and Wing [30] use this model to define a correctness condition called *linearizability* that extends Lamport's notion of atomicity for reads and writes [36] to arbitrary data

types. Linearizability requires that in any (concurrent) execution, each operation "appears" to take effect instantaneously, sometime between the invocation and response events of the operation. Linearizability is also similar to Lamport's *sequential consistency* [35], but requires that if two operations on a given object do not overlap in time, then the order in which they "appear" to occur is consistent with the order in which they actually occur. And unlike sequential consistency, linearizability has a locality property: if each object is linearizable, then the entire system is linearizable.

In this section, we take the linearizability notion one step further, and show that a linearizable invocation-response system is equivalent to an atomic access system. In particular, we show that if the objects in the invocation-response system are each *linearizable*, then every behavior of the entire invocation-response system is a behavior of the atomic access system. Thus, in reasoning about complex systems, it is possible to consider overlapping operation executions at one level of abstraction, and shared atomic actions for the same operations when reasoning at higher levels of abstraction. This is an important benefit of using a model that unifies invocation-response and atomic access in a single formal framework. In addition, we extend the work of Herlihy and Wing by treating not only safety properties of invocation-response systems, but liveness properties as well.

These results are intended to be used for reasoning about multiprocessor programs, in which the shared memory is assumed to be atomic and the program is modularized by layers of linearizable concurrent objects. They may also be useful in showing that memory systems themselves are atomic, e.g., complex cache-coherence algorithms could be modeled explicitly, and a proof that such a system is linearizable would justify claims that the programmer can ignore details of the memory system.

We begin, in Section 7.3.1 by describing the basic architecture of an invocation-response system, including the interfaces and specification mechanisms for the objects and processes. Then, in Section 7.3.2, we describe three systems: a concurrent system, a sequential system, and an atomic system. All three systems are described in the I/O automaton model. We show that if the objects of the concurrent system are linearizable and if the processes obey certain well-formedness restrictions, then each of these systems "simulates" the next. This gives us a unified theory for describing systems in terms of invocations and responses, but reasoning

about them in terms of atomic accesses.

7.3.1 Invocation-Response Systems

We are interested in studying systems in which processes invoke operations on objects and then wait for the objects to respond. In this section, we define a general architecture for *invocation-response* systems (or *IR* systems). Later, this architecture will be used to define two systems: System C, a concurrent system containing linearizable objects, and System B, a sequential system used as a stepping stone in our proof. A different structure will be used to define System A, an atomic shared memory system that will form the basis of our correctness condition.

An IR system consists of a set of processes and a set of objects, where each process and each object is modelled as an I/O automaton. Processes may request operations on objects by issuing "invoke" actions. These actions are inputs to the objects, which issue "respond" output actions after performing the requested operation. The interface at the boundary between a process and its environment is analogous to the interface at the boundary between an object and a process. To request that the system perform a particular function, the environment may "invoke" operations on a process, which later replies to the environment with a "respond" action. Here, we consider systems of only three layers: the objects, the processes, and the environment. However, by modelling the interaction between a process and the environment in the same way as the interaction between an object and a process, we set the stage for constructing complicated objects hierarchically. That is, one might compose a collection of objects and processes, and treat the composition as a single object. To describe the set of operations that may be invoked on an object or process, we define an "interface type." An *interface type* T consists of:

- $ops(T)$, a set of operation names, and
- for each operation $\rho \in ops(T)$,
  - $args(\rho)$, the domain of arguments to the operation, and
  - $rets(\rho)$, the domain of return values of the operation.

The operation names identify the operations that may be invoked on the corresponding object or process. For each operation name, the domain of argument values specifies the allowable operation arguments that may be supplied by the user of the object. Similarly, the return value domain for an operation specifies the possible values that may be returned by the object as a result of that operation. We will see shortly how the interface type of an object or process is used to derive the signature of the corresponding automaton.

In IR systems, there are three kinds of components: the shared objects, the processes that invoke operations on those objects, and the environment that directs the activities of the processes. For the remainder of the chapter, we fix three sets of indices, $\mathcal{I}$, $\mathcal{J}$, and $\mathcal{K}$. We use the elements of $\mathcal{I}$ to name the objects in a system, and we use the elements of $\mathcal{J}$ to name the processes. An IR system is modelled as the composition of an object automaton o_i for each $i \in \mathcal{I}$, and a process automaton p_j for each $j \in \mathcal{J}$. The indices in $\mathcal{K}$ identify the processes (or users) that constitute the environment of a system. We do not model the environment explicitly, but simply use the elements of $\mathcal{K}$ to refer to its components. One may think of these components either as I/O automata or as users interacting with the system. We require that the environment, as a whole, obey certain well-formedness restrictions on its interactions with each process. Informally, we require that the environment wait for a process to respond to a request before making a new request of that process.[6] If several components in the environment may make requests of the same process, then those components must cooperate (possibly by participating in a mutual exclusion protocol) in order to ensure that well-formedness is preserved at that process. We now define the objects and processes of IR systems.

Objects

Each object automaton $o_i, i \in \mathcal{I}$, has an associated interface type, denoted $type(i)$, and the signature of o_i is determined from this interface type. For each $i \in \mathcal{I}$, we define $sig(o_i)$ as follows:

Input Actions: $\text{invoke}_{i,j}(\rho, a)$, $j \in \mathcal{J}$, $\rho \in ops(type(i))$ and $a \in args(\rho)$

Output Actions: $\text{respond}_{i,j}(\rho, a, r)$, $j \in \mathcal{J}$, $\rho \in ops(type(i))$, $a \in args(\rho)$, and $r \in rets(\rho)$

The subscripts on each action identify the object automaton o_i at which the operation occurs

[6] A similar idea appears in [45] on page 79.

and the process automaton p_j responsible for the request. The following definition is useful for describing executions of o_i. Let α be an execution of o_i. We say that α is *input well-formed* iff $\forall j \in \mathcal{J}$, no two $\text{invoke}_{i,j}$ actions occur in α without a $\text{respond}_{i,j}$ action between them.

Processes

Each process in an IR system is modelled as an I/O automaton p_j, $j \in \mathcal{J}$, that has an associated interface type, denoted $type(j)$. The interface type describes the set of operations that may be invoked on a process. In addition, a process may itself invoke operations on objects. Therefore, its signature not only contains actions corresponding to operations in its interface type, but also invoke and respond actions corresponding to the interface types of the objects that it may access. For each $j \in \mathcal{J}$, we let $obj(j) \subseteq \mathcal{I}$ denote that set of objects that p_j may access, and define the signature of p_j as follows:

Input Actions: $\text{invoke}_{j,k}(\rho, a)$, where $k \in \mathcal{K}, \rho \in ops(type(j))$ and $a \in args(\rho)$

$\text{respond}_{i,j}(\rho, a, r)$, where $i \in obj(j), \rho \in ops(type(i))$, $a \in args(\rho)$, and $r \in rets(\rho)$

Output Actions: $\text{respond}_{j,k}(\rho, a, r)$, where $k \in \mathcal{K}, \rho \in ops(type(j))$, $a \in args(\rho)$, and $r \in rets(\rho)$

$\text{invoke}_{i,j}(\rho, a)$, where $i \in obj(j), \rho \in ops(type(i))$ and $a \in args(\rho)$

In reasoning about the schedules of a process in an IR system, it will be helpful to distinguish those actions that are shared with the objects from those shared with the environment. Let β be a sequence of actions of p_j. We define $\beta|\mathcal{I}$ to be the subsequence of β containing exactly the $\text{invoke}_{i,j}$ and $\text{respond}_{i,j}$ actions, for all $i \in \mathcal{I}$. Similarly, we define $\beta|\mathcal{K}$ to be the subsequence of β containing exactly the $\text{invoke}_{j,k}$ and $\text{respond}_{j,k}$ actions, for all $k \in \mathcal{K}$.

As mentioned earlier, we constrain the interaction between a each process and the environment so that a process receives no inputs from the environment while the process has an outstanding request. That is, we want the invocations and responses at the environment boundary of each process to alternate, where each response is appropriate for the preceding invocation. For this purpose, we use the following definition. If γ is a sequence of actions, $j \in \mathcal{J}$, and $\beta = \gamma|p_j$, we say that γ is *externally well-formed* for j iff $\beta|\mathcal{K}$ is an alter-

nating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall k \in \mathcal{K}, \forall \rho \in ops(P_j), \forall a \in args(\rho), \forall r \in rets(\rho)$, each $\text{respond}_{j,k}(\rho, a, r)$ action is immediately preceded by an $\text{invoke}_{j,k}(\rho, a)$ action. We say that γ is externally well-formed iff it is externally well-formed for all $j \in \mathcal{J}$. An execution is externally well-formed iff its schedule is well-formed.

In externally well-formed sequences, we think of a process as being "active" in the interval between receiving an invocation and generating a response. More formally, let β be an externally-well formed sequence of p_j, $j \in \mathcal{J}$. If β' is a prefix of β, we say that p_j is *active* after β' iff $\beta'|K$ ends with an invoke action. It is important to notice that in externally well-formed executions, a process receives no inputs from the environment while it is active.

The processes in an IR system model the specific application program that accesses the objects. Therefore, in stating the general definition of an IR system, we do not explicitly define the process automata. However, we do require that each p_j, $j \in \mathcal{J}$ preserves the following well-formedness condition. Let β be a sequence of actions, and let $\beta_j = \beta|p_j$, $j \in \mathcal{J}$. We say that β is *well-formed* for j iff the following conditions hold:

- β is externally well-formed for j.
- Every action in $\beta_j|\mathcal{I}$ occurs from a prefix of β_j after which p_j is active.
- The sequence $\beta_j|\mathcal{I}$ is an alternating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall i \in \mathcal{I}, \forall \rho \in ops(type(i)), \forall a \in args(\rho), \forall r \in rets(\rho)$, each $\text{respond}_{i,j}(\rho, a, r)$ action is immediately preceded in β_j by an $\text{invoke}_{i,j}(\rho, a)$ action.

So, in order to preserve well-formedness, p_j must respond at most once to each request from the environment, and it may invoke operations on objects only in the interval between a request from the environment and its response to that request. Furthermore, if p_j invokes an operation on an object, it may not produce any output actions until the corresponding response from that object occurs. Note that the third condition implies that p_j preserves input well-formedness in β for all $i \in \mathcal{I}$.

7.3.2 Simulating Atomic Access Systems with IR Systems

At the beginning of Section 7.3, we stated that an important problem in programming multiprocessor systems is to build IR systems containing concurrently accessed shared objects in

such a way that the environment cannot distinguish them from atomically accessed shared memory systems. In this section, we take advantage of ability to study both shared memory and message-passing systems in the I/O automaton model in order to show a formal correspondence between systems containing concurrently accessed linearizable objects and systems having atomically accessed shared memory. We present three systems. The first is an IR system (System C) that models the system containing concurrently accessed linearizable objects. The second system, derived from the first, is an IR system (System B) in which at most one operation is in progress at each object at any time. Finally, we present an atomic access system (System A) that corresponds to system B, but implements the objects in atomically accessed shared memory. We show that the fair behaviors of System C are contained in those of System B, and that the fair behaviors of System B are contained in those of System A. This serves to formalize the notion that systems containing linearizable objects "simulate" those in which the objects are implemented in atomic memory. We begin with an overview of the three systems.

System C is the concurrent invocation-response system that we wish to prove simulates an atomic memory system. It is an IR system, so each process of System C has an interface type, the appropriate signature for that type, and is required to preserve well-formedness. In addition, the processes must satisfy a property that implies that objects eventually respond to all operation requests. In order to define the objects of System C, we present a natural definition for a "sequential specification" of an object and define formally what it means for an object to be a "linearizable implementation" of such a specification. We require that each object of System C is described by a sequential specification and is constrained to be a linearizable implementation of that specification. But aside from the above restrictions, the processes and objects of System C are completely general. We do not use any information about the particular sequential specifications or implementations of the objects in order to prove our results. In this way, our results hold for any IR system with linearizable objects.

As we have said, our notion of correctness is that every fair behavior of System C should be a fair behavior of a system in which the objects are accessed atomically (as opposed to separate invocations and responses). In other words, System C should "simulate" a system in which the objects are implemented in an atomic shared memory. Rather than showing this simulation directly, we construct an intermediate system in which the processes are the same as in System

C, but in which the objects are constructed explicitly from their sequential specifications. This intermediate system, called System B, is used as a stepping stone in the proof. We show that for every fair execution of System C, there is a fair execution of System B having the same external behavior, and in which each invocation of an operation on an object is immediately followed by the corresponding response.

Finally, we construct System A, the atomic system that forms the basis of our correctness condition. System A consists of a set of processes that perform atomic accesses on a shared memory. The system is constructed from the processes of System C and the sequential specifications of the objects. We show that for every fair execution of System B in which invocations are immediately followed by their corresponding responses, there is a fair execution of system A with the same behavior. The two simulation arguments (that System C simulates a certain class of executions of System B, and that those executions correspond to executions of System A) are combined to complete the proof.

System C

In this section, we define System C, the IR system that we wish to prove correct. System C is the composition of a collection of linearizable objects and processes that we wish to show behaves correctly. The automata in System C are not given to us explicitly, but are guaranteed to satisfy certain properties. From the definition of an IR system, we know that in System C each process automaton p_j, $j \in \mathcal{J}$, preserves well-formedness. Furthermore, we will assume that each object in System C is a "linearizable implementation" of a "sequential specification". In addition, we will make an assumption about liveness in System C. We begin by defining a sequential specification, and say what it means to be a linearizable implementation of one.

For each $i \in \mathcal{I}$, we fix a *sequential specification* S_i consisting of the following information:

- $states(i)$, a set of states containing a set of initial states $init(i)$,
- two predicates for each operation name $\rho \in ops(type(i))$:
  - predicate P_ρ on elements from $args(\rho) \times states(i)$, and
  - predicate Q_ρ on elements from $args(\rho) \times states(i) \times states(i) \times rets(\rho)$.

This means that if $a \in args(\rho)$ is the argument to operation ρ, $x' \in states(i)$ is the "current state" of object O, and the predicate P_ρ holds on a and x', then there exists an $x \in states(i)$ and an $r \in rets(\rho)$ such that $Q_\rho(a, x', x, r)$ is true. The value x becomes the current state of O following the operation, and r is returned by the operation. For arguments a and states x' for which $P_\rho(a, x')$ does not hold, the new state and return value are unspecified. In Larch specifications[8], this information is conveniently represented in the following way:

ρ = proc(a:$args(\rho)$) returns (r:$rets(\rho)$)
pre: $P_\rho(a, x')$
effect: $Q_\rho(a, x', x, r)$

Having defined a sequential specification, we now wish to define what it means for the object automata of System C to be linearizable implementations of their sequential specifications.

Borrowing a technique from [41], we construct a particular automaton, called a "sequential object" that captures the meaning of a sequential specification. The sequential object construction will be used not only to define a linearizable implementation of a sequential specification, but also to define System B.

We capture the meaning of each sequential specification S_i, $i \in \mathcal{I}$, with a *sequential object automaton* a_i. The sequential object automaton a_i has signature $sig(o_i)$ and the following state components: *current* $\in states(i)$, *user* $\in \mathcal{J} \cup \perp$, *op* $\in ops(type(i)) \cup \perp$, and *arg* $\in \bigcup_{\rho \in ops(type(i))} args(\rho) \cup \perp$. The component *current* holds the "current state" of the object, and is initially in $init(i)$. The component *user*, initially $\perp$, is the index of the process currently using the object. Components *op* and *arg* hold the name and argument of the operation in progress; initially, these are both $\perp$. The transition relation for the sequential object automaton is given in Figure 7-2. The partition of a_i consists of a single class containing all the output actions of a_i.

When an invocation occurs at a_i, the automaton simply stores the id of the process making the request, the name of the operation, and the values of the arguments to the operation. Whenever an operation has been requested but the response has not yet occurred, a_i may respond to the request, supplying a return value consistent with the sequential specification and resetting the *user*, *op* and *arg* components to their initial values.

- $\text{invoke}_{i,j}(\rho, a)$
  Eff: $s.user = j$
  $s.op = \rho$
  $s.arg = a$

- $\text{respond}_{i,j}(\rho, a, r)$
  Pre: $s'.user = j$
  $s'.op = \rho$
  $s'.arg = a$
  $P_\rho(a, s'.current)$
  Eff: $Q_\rho(a, s'.current, s.current, r)$
  $s.user = \perp$
  $s.op = \perp$
  $s.arg = \perp$

Figure 7-2: Transition relation for sequential object automaton a_i.

Next, we would like to define what it means for an automaton to be a linearizable implementation of sequential specification S_i. But first we need to define a particular class of executions of a_i. We say that α is a *sequential execution* of a_i iff $sched(\alpha)$ is an alternating sequence of request and respond actions.

We can now define linearizability. We say that o_i is a *linearizable implementation of* S_i iff for all input well-formed fair executions α of o_i, there exists a sequential fair execution α' of a_i such that for all $j, j' \in \mathcal{J}$,

1. $\alpha'|p_j = \alpha|p_j$, and

2. if a $\text{respond}_{i,j'}$ event π' precedes an $\text{invoke}_{i,j}$ event π in α, then π' precedes π in α'.

Informally, the first condition says that each individual process cannot distinguish α from α'. The second condition says that if the invocation-response intervals of two operations at an object do not overlap in α, then they must occur in the same relative order in α' as they do in α.

For every $i \in \mathcal{I}$, we require that o_i in System C is a *linearizable implementation of* S_i. Note that the linearizable implementation requirement is a local property of each object, and not a property of the system as a whole. We will consider global linearizability properties after defining System B.

Our final assumption about System C concerns liveness. We require that all externally well-formed executions γ of System C are *response-live*, meaning that for all $i \in \mathcal{I}$, for all $j \in \mathcal{J}$, for all $\rho \in ops(type(i))$, for all $a \in args(\rho)$, if $\pi = invoke_{i,j}(\rho, a)$ occurs in γ, then there exists a state s after π such that some $respond_{i,j}$ action is enabled from each state after s until such an action occurs. With this assumption, we get the following liveness result:

Lemma 7.25: Let γ' be an externally well-formed fair execution of System C. Then for all $i \in \mathcal{I}$, for all $j \in \mathcal{J}$, if an $invoke_{i,j}$ action occurs in γ then a $respond_{i,j}$ action occurs later in γ.

Proof: Immediate from the definitions of response-live and fairness. ■

Notice that we could have imposed a condition stronger than response-liveness by prohibiting partial operations in sequential specifications entirely. (Prohibiting partial operations would ensure, by the definition of a_i and linearizability, that each object eventually responds to each request.) However, in order to allow modelling a class of systems in which the processes cooperate to ensure that operations are invoked only when appropriate, we choose to take a more general approach, in which the system must guarantee that a response eventually occurs for each request. For example, an object might be responsible for granting permission to use a shared resource (a lock) so that no two processes have permission simultaneously. Such an object could have two operations, one for requesting the lock and another for releasing it. If one process requests the lock while a second process process is holding the lock, then the object cannot respond to the request until the second process releases the lock. Thus, the operation is partial, but as long as processes are guaranteed to eventually release the lock, then all requests can be satisfied.

A special case of the response-live property is one that says that for all $i \in \mathcal{I}$, for all $j \in \mathcal{J}$, for all $\rho \in ops(type(i))$, for all $a \in args(\rho)$, if $\pi = invoke_{i,j}(\rho, a)$ occurs in γ, then some $respond_{i,j}$ action is enabled from each state after π until such an action occurs. In other words, if an object has partial operations, then (1) they are invoked only in states for which they are defined, and (2) if an operation is pending at a process, then no state change occurs to prevent a response to that operation. Although this property is stronger than the response-liveness property, it is a *safety* property and may be easier to prove when it is applicable.

System B

Rather than directly showing that System C simulates an atomic access system, it will be convenient to define an intermediate system, System B. We define System B to be identical to System C except that for all $i \in \mathcal{I}$, each object automaton o_i is replaced by a_i, the sequential object automaton corresponding to the sequential specification S_i.

Let β be an execution of System B. We say that β is a *sequential execution* of System B iff for all $i \in \mathcal{I}$, $\beta|a_i$ is a sequential execution of a_i and no actions occur in β between each request/response pair of $\beta|a_i$. So, $\beta|\mathcal{I}$ consists of an alternating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \forall \rho \in ops(type(i)), \forall a \in args(\rho), \forall r \in rets(\rho)$, each $\text{respond}_{i,j}(\rho, a, r)$ action is immediately preceded by an $\text{invoke}_{i,j}(\rho, a)$ action. We now prove our first simulation result.

Lemma 7.26: Let γ be an externally well-formed fair execution of System C. Then there exists a sequential fair execution β of System B such that for all $j \in \mathcal{J}, \beta|p_j = \gamma|p_j$.

Proof:[7] From the definition of System C, all processes p_j, $j \in \mathcal{J}$ and objects o_i, $i \in \mathcal{I}$ preserve well-formedness. Therefore, since γ is externally well-formed, we know that for all $j \in \mathcal{J}$, $\gamma|p_j$ is well-formed. Recall this means that:

- Every action in $\gamma_j|\mathcal{I}$ occurs from a prefix of γ_j after which p_j is active.
- The sequence $\gamma_j|\mathcal{I}$ is an alternating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall i \in \mathcal{I}, \forall p \in ops(type(i)), \forall a \in args(p), \forall r \in rets(p)$, each $\text{respond}_{i,j}(p, a, r)$ action is immediately preceded in γ_j by an $\text{invoke}_{i,j}(p, a)$ action.

The second condition induces a total order $<_j$ on the operations invoked by p_j, and by Lemma 7.25, we know that each invocation has a matching response. Furthermore, since each object in System C is linearizable, we know that for each $i \in \mathcal{I}$, we can fix a fair execution γ_i of a_i such that for all $j, j' \in \mathcal{J}$,

1. $\gamma_i|p_j = (\gamma|o_i)|p_j$, and
2. if a $\text{respond}_{i,j'}$ event π' precedes an $\text{invoke}_{i,j}$ event π in γ, then π' precedes π in γ_i.

[7] This proof follows closely the proof of a similar theorem in [30] and uses ideas from [45], page 78, to treat the actions of the environment.

Each γ_i induces a total order $<_i$ on the operations invoked in o_i in γ.

In order to show that the execution β exists, we first show that there exists a total order $<_T$ on all the operations invoked in γ that is consistent with all the total orders $<_j, j \in \mathcal{J}$ and $<_i, i \in \mathcal{I}$. It is sufficient to show that the transitive closure $\prec$ of the union of all $<_j, j \in \mathcal{J}$ and $<_i, i \in \mathcal{I}$ is a partial order. Suppose not. Then there exists some cycle in $\prec$. We know that the cycle must involve a pair of operations ordered by $<_j$ for some $j \in \mathcal{J}$. Otherwise, all the operations in the cycle would be ordered by the *same* $<_i, i \in \mathcal{I}$, an immediate contradiction, since $<_i$ is a total order. Let op_1 and op_2 denote the two operations ordered by $<_j$ in our cycle, and without loss of generality, let $op_1 <_j op_2$. This means that the response for op_1 occurs in γ before the invocation of op_2. Since op_1 and op_2 are in a cycle, we also know that there exists a sequence of operations $op_1, op_2, \ldots, op_n$ with $op_1 = op_n$ such that the response of each op_m precedes the invocation of op_{m+1}. But this means that the response of op_1 precedes the invocation of op_1 in γ, a contradiction.

So, to construct β, we first construct the schedule of β by taking the sequence of the invocation/response pairs in the order specified by $<_T$, and then, for all $j \in \mathcal{J}$, $k \in \mathcal{K}$, inserting in all $invoke_{j,k}$ and $respond_{j,k}$ actions appearing in γ so that $\beta|p_j = \gamma|p_j$. That is, we place each invocation/response pair of the environment "around" the corresponding sequence of object operations pairs, and place any invocation from the environment that is lacking a response after the last object operation pair. Now, we know from the construction of $<_T$ that for all $i \in \mathcal{I}$, $sched(\beta)|o_i = sched(\gamma)|o_i$. Furthermore, by the construction of $<_T$ and the alternating sequence condition of well-formedness for j, we know that for all $j \in \mathcal{J}$, $(sched(\beta)|\mathcal{I})|p_j = (sched(\gamma)|\mathcal{I})|p_j$. And from well-formedness, we know that for all $j \in \mathcal{J}$, every action in $\gamma_j|\mathcal{I}$ occurs from a prefix of γ_j after which p_j is active. Therefore, we know that it is possible to place the invocation/response pairs of the environment around the corresponding sequence invocation/response object operation pairs so that for all $j \in \mathcal{J}$, $sched(\beta)|p_j = sched(\gamma)|p_j$. Now, since all processes and objects have the same schedules in β as in γ, we can insert the states of β so that for all $j \in \mathcal{J}$, the sequence of state transitions in β for p_j is the same in β as in γ. (In other words, because its schedule is the same, each process p_j cannot tell whether it is in β or in γ.) Since each object o_i is a linearizable implementation of its sequential specification, we know that for all $i \in \mathcal{I}$, there exists a sequential fair execution β_i of a_i with

schedule $sched(\gamma)|o_i$. Therefore, for each $i \in \mathcal{I}$, we let the sequence of state transitions of a_i in β be as in β_i. For each object a_i, we know that a response occurs in β for each invocation, so $\beta|a_i$ is fair. Since γ is fair and for all $j \in \mathcal{J}$, $\beta|p_j = \gamma|p_j$, we know that $\beta|p_j$ is fair, for all j. So, applying Lemma 2.1, we know that β is a sequential fair execution of System B. ■

This result tells us that any externally well-formed fair execution of System C looks to the environment as if it is a sequential fair execution of System B. Now, we would like to say that any sequential fair execution of System B looks to the environment as if it is a fair execution of a system in which the objects are implemented in atomically accessed shared memory. This brings us to System A.

System A

In this section, we define System A, which forms the basis of our correctness condition. System A is a system in which objects are modelled as variables in a global shared memory that is accessed atomically by the processes. It is in the construction of System A (and the related proofs) that we exploit the shared memory extensions of the I/O automaton model. They allow us to model and reason about both the atomic access systems and the IR systems using a single unified model.

In order to define System A, we need a general transformation that takes a process automaton (as given to us in System C) and a set of sequential specifications (also given), and produces a shared memory automaton that corresponds to the original process but accesses the objects as atomic variables in a shared memory. We now define this transformation.

Given a process automaton p_j that accesses shared variables using the invocation-response mechanism as described above, we can construct an "equivalent" shared memory automaton s_j that accesses shared variables using atomic accesses to a shared memory. Since the transition relation of the shared memory automaton must specify how the shared variables are updated, the definition of s_j depends not only upon the definition of p_j, but also upon the sequential specifications for the objects that p_j accesses.

In order to change the style of variable access, we need to replace the invoke and response actions in the signature by atomic shared memory actions. For all $i \in \mathcal{I}$, we let X_i denote $states(i)$, $X = \bigcup_{i \in \mathcal{I}} X_i$, and $\bar{X}_i = X \setminus \{X_i\}$. Automaton s_j is defined as follows.

- $sig(s_j)$ is the signature:

  Input Actions: $\text{invoke}_{j,k}(\rho, a)$, where $k \in \mathcal{K}, \rho \in ops(P_j)$ and $a \in args(\rho)$

  Output Actions: $\text{respond}_{j,k}(\rho, a, r)$, where $k \in \mathcal{K}, \rho \in ops(P_j)$, $a \in args(\rho)$, and $r \in rets(\rho)$

  $(v', \rho_{i,j}(a), v)$, where $v, v' \in dom(X)$, $i \in \mathcal{I}$, $\rho \in ops(type(i))$, and $a \in args(\rho)$.

- $states(s_j) = states(p_j)$,

- $start(s_j) = start(p_j)$,

- $steps(s_j)$ = the set of all steps (s'', π, s) such that either

  1. $\pi = \text{invoke}_{j,k}$ or $\text{respond}_{j,k}$, $k \in \mathcal{K}$ and $(s'', \pi, s) \in steps(p_i)$, or
  2. $\pi = (v', \rho_{i,j}(a), v), s)$ and $\exists r, s'$ such that
     (a) $(s'', \text{invoke}_{i,j}(\rho, a), s') \in steps(p_i)$,
     (b) $(s', \text{respond}_{i,j}(\rho, a, r), s) \in steps(p_i)$,
     (c) $P_\rho(a, v'|X_i) \Rightarrow Q_\rho(a, v'|X_i, v|X_i, r)$, and
     (d) $v|\bar{X}_i = v'|\bar{X}_i$.

- $part(s_j) = part(p_j)$, except that each $\text{request}_{i,j}(\rho, a)$ action is replaced the actions $(v', \rho_{i,j}(a), v)$.

A few words explaining the transition relation for s_j are in order. We include directly in the steps of s_j each step of p_j for a process invocation or a response to the environment (i.e., each step not involving an object access). In addition, we include shared memory steps that correspond to invocation/response pairs for objects in System B. Conditions (a) and (b) say that the state change that occurs at s_j as a result of the atomic access corresponds to a state change that can occur in p_j as a result of the invocation/response pair. Condition (c) ensures that the new value of the shared variable X_i is consistent with the sequential specification S_i. Finally, condition (d) says that no shared variables other than X_i are changed by the step.

Lemma 7.27: For all $j \in \mathcal{J}$, s_j is a shared memory automaton for X.

Proof: Immediate from the definitions of I/O automata and shared memory automata. ■

We define System A to be the composition of the shared memory automata corresponding to the process automata of System C, closed out on the entire set of shared variables. More formally, $C = C(\Pi_{j \in \mathcal{J}} s_j, X)$.

Lemma 7.28: Let β be a sequential fair execution of System B. Then there exists a fair execution α of System A such that $beh(\alpha) = beh(\beta)$.

Proof: We "collapse" β to get α: Since β is a sequential execution, each object operation invocation is immediately followed by its corresponding response. Therefore, to construct α, for all $i \in \mathcal{I}$, $j \in \mathcal{J}$, we replace each subsequence

$$s'', invoke_{i,j}(\rho, a), s', respond_{i,j}(\rho, a, r), s$$

in β by the corresponding step

$$(\hat{s''}, \rho_{i,j}(a), \hat{s}) \in steps(A)$$

in α such that for all $j' \in \mathcal{J}$ $\hat{s''}|s_{j'} = s''|p_{j'}$, $\hat{s}|s_{j'} = s|p_{j'}$, and for all $i' \in \mathcal{I}$, the values of $X_{i'}$ in $\hat{s''}$ and $\hat{s}$ match $(s''|a_{i'}).current$ and $(s|a_{i'}).current$, respectively. From the definition of a_i that the state change at a_i is between s'' and s is consistent with the sequential specification S_i. Therefore, we know that the step $(\hat{s''}, \rho(a), \hat{s})$ must exist in $steps$(A). By the definition of s_j, α is an execution of A. To see that α is fair, we note that β is fair and each action $(v', \rho_{i,j}(a), v)$ of s_j is enabled exactly from those states in which $request_{i,j}(\rho, a)$ is enabled in p_j. ■

Our main result follows immediately.

Theorem 7.29: Let γ be a fair execution of System C. Then there exists a fair execution α of System A such that $beh(\alpha) = beh(\gamma)$.

Proof: Immediate from Lemmas 7.26 and 7.28. ■

Thus, we have shown formally that if the objects in an invocation-response system are each linearizable, then every fair behavior of the entire invocation-response system is a fair behavior of the corresponding atomic access system.

```
AUTOMATON mutex
   STATE  tuple(stage:string, checked:set(automaton_id), k: automaton_id)
   SHARED tuple(control:mapping(automaton_id,integer), k: automaton_id)
   CLASS
      OUTPUT Read
         PRE eq(s.stage,"read")
         EFF s.k = v.k
             ifthenelse(eq(s.k, self()),
                        assign(s.stage,"control2"),
                        assign(s.stage,"check"))
```

Figure 7-3: Automaton type definition for a shared memory automaton.

7.4 Supporting Shared Memory in Spectrum

Although Spectrum does not currently provide support for the shared memory model extensions described above, we were able to use Spectrum to simulate the example algorithm presented in Section 7.2 *by explicitly constructing the closed out automaton.* The invariants and variant function were mechanically checked for random executions of the algorithm. In particular, Lemma 7.17 was checked mechanically for all states of random executions of the algorithm. Furthermore, Lemma 7.21 (and earlier incorrect versions of it) was checked for random algorithm executions. That is, for each step it was mechanically verified that either (1) progress was being made (see Lemma 7.20), or (2) the variant function decreased, or (3) the variant function was unchanged and one of the exceptions held.

However, all of this would have been simpler had Spectrum provided shared memory automata and a closeout operator. In this section, we suggest extensions to Spectrum that would permit one to express and simulate shared memory algorithms.

Our first problem is to extend the Spectrum language to allow shared variables to be declared in automaton type definitions. For this, we provide a new keyword, SHARED, that has the same syntax as STATE declarations, but declares the names and types of shared variables. The granularity of sharing is at the level of the components of the top level tuple. Figure 7-3 illustrates the syntax for shared memory automaton type definitions, and corresponds to a part of the shared memory mutual exclusion algorithm shown in Figure 7-1. The shared variables declared are `control` and `k`. In order to enforce the property of shared memory automata

that an automaton must be prepared to observe any possible value in the shared memory, we require that shared variables appear only in EFF clauses. Furthermore, we require that shared variables are not modified in the EFF clauses of input actions, and a mechanism[8] is needed to ensure that shared memory input actions observe the "prior" states of the shared variables. The shared variables are referenced using the notation `v.control` and `v.k`, as in the EFF clause of the read action shown, and we may close out on these separately if we like.

Just as for composition, we place the closeout operator in the user interface. First, we must allow the user interface to call procedures in the loader for type checking purposes. Whenever we add a new shared memory automaton type to a composed type, we must check that if it has a shared variable declared with a given name, then either (1) none of the other automaton types in the composition have a shared variable with that name, or (2) the corresponding variable in the composition has the same data type. We modify the auxiliary windows for composed types so that any shared variables are listed with their data types at the top of the window. To close out on a particular variable in a composed type, one simply selects that variable in the window. Repeatedly selecting the variable toggles between closing out on that variable or not.[9] Those variables on which one has closed out are highlighted in a different color.

Finally, changes are needed in the simulator for managing the shared variables. The loader must be modified to recognize the SHARED keyword and the `v.` notation, perform the related type checking for shared variables, and enforce the requirement that shared variables are referenced in EFF clauses only. In addition, the interpreter must keep, for each closed out automaton, the shared variables associated with that automaton and their values. The execution loop and scheduler are unaffected.

7.5 Summary

In this chapter, we extended the I/O automaton model to allow modelling of shared memory systems, as well as systems that include both shared memory and shared action communication. The extended model was shown to support all types of atomic accesses to shared memory, from

[8] Such a mechanism might be saving a temporary copy of the shared variables, or saving for last the evaluation the state transition for the automaton generating the output.

[9] Note that if a composed type is a component of yet another composed type, then deciding not to close out on a variable would require type checking of the shared variables at the next level up to ensure compatibility.

the very restrictive single-variable reads and writes to operations on arbitrary abstract data types. By building our shared memory model on top of I/O automata, we could take advantage of the fairness definitions and compositionality properties already present in that model. This resulted in a unified model with relatively few new concepts. An example algorithm, Dijkstra's classical shared memory mutual exclusion algorithm, was presented in this model and its safety and progress properties were shown using standard assertional and variant function techniques. Then, using the extended model, we showed a formal correspondence between systems containing linearizable objects and systems containing atomically accessed shared variables. Finally, we proposed extensions to the Spectrum Simulation System for expressing and simulating shared memory algorithms.

Chapter 8

Superposition

Modular descriptions of distributed algorithms are sometimes most easily written in terms of several program layers. Higher layers are allowed to make use of lower layers, but lower layers are unaware of the higher layers. One layering mechanism, called *superposition*, is defined by Chandy and Misra for the UNITY programming language [14]. A UNITY program consists of a set of *statements* that access a global shared memory. At each step in the execution, a statement is selected and executed, possibly updating the memory. Superposition in UNITY is defined to be a program transformation that adds a layer on top of a program, while preserving all the properties of the underlying program. Essentially, the transformation modifies the underlying program by adding a set of new variables and some extra code to make use of them. In order to preserve the properties of the underlying program, the extra code does not modify the original variables (although it may read them). Unfortunately, modularity is lacking in UNITY because the interfaces between program modules are not described in terms of well-defined sets of actions, but only in terms of the program variables that they access. Therefore, one must reason about programs not in terms of actions that occur at module boundaries, but in terms of the memory locations that modules read and write. That is, one cannot treat a module as an abstraction with a certain set of behaviors, but must must always be concerned with internal state the module. In addition, UNITY has no notion of an action being an output of one component and an input to another. We would like such a separation for describing distributed systems.

Partly because of its separation of inputs and outputs, the I/O automaton model is partic-

ularly natural for describing distributed systems. It permits writing precise problem specifications, clear algorithm descriptions, and careful correctness proofs. Unlike UNITY, communication in this model takes place entirely in terms of actions shared across module boundaries. Each module has its own local state variables, unseen by other modules. The compositionality results of the model make it possible to reason locally about system components in order to prove properties about executions of the entire system. However, the I/O automaton model does not provide a mechanism for constructing layered systems in which higher level modules can observe the states of lower level ones. Thus, UNITY has a superposition mechanism but little modularity, while I/O automata provide a great deal of modularity but no superposition mechanism.

In this chapter, the I/O automaton model is extended to permit superposition of program modules. Rather than viewing superposition as a program transformation, we view it as a particular method for hierarchically combining separate program modules. When one module is superposed on another, the higher level module is allowed to observe (but not modify) the state of the underlying module, while the state of the higher level is unknown to the underlying module. We define an operator for superposing one I/O automaton on another, and show that superposition does not affect the set of executions of the underlying module, thus preserving all properties of that module. A formal specification mechanism is presented that allows the set of correct behaviors of the higher level module to be expressed in terms of the state of the underlying module. As an illustration of the extended model, the global snapshot algorithm of Chandy and Lamport [13] is presented with a complete proof of correctness.

A different approach to adding superposition to the I/O automaton model is presented by Nour [51]. In that work, a restricted class of I/O automata, called UNITY automata, is defined in order to express UNITY programs as I/O automata. A superposition operator is defined for this restricted class. Since UNITY automata are restricted to have output actions only, it is not possible to model a superposition in which the higher level module may share actions with the lower level module. In the present work, we do not need such restrictions. In fact, our example algorithm makes important use of shared actions between layers.

In the preceding chapter, we extended the I/O automaton model in order to permit automata to make make atomic accesses to shared variables. The variables were modelled as being

completely external to the automata sharing them, so an automaton had to be prepared to observe any value in the memory whenever it executes an access. In this chapter, variables are shared, but the sharing relationship is different. The higher level module sees the variables of the lower level module *at all times*. It is not necessary for the higher level automaton to execute a particular action in order to observe the values of those variables. Therefore, the set of actions "enabled" in the higher level module may change as the lower level module updates its variables. This sort of relationship cannot be modelled using the shared memory extensions of Chapter 7.

The remainder of the chapter is organized as follows. In Section 8.1, we define the superposition extensions and prove several properties of the extended model. This is followed by the global snapshot example in Section 8.2. In Section 8.3, we propose extensions to the Spectrum Simulation System to support superposition.

8.1 Superposition Extensions

In this section, we present definitions that extend the I/O automaton model for superposition of program modules. We begin by defining what it means for an automaton to be "unconstrained" for a particular set of variables, and use this definition to state the requirements for one automaton to be "superposable" on another. We then define the superposition operator, and show that the superposition of one I/O automaton on another produces a new I/O automaton. Therefore, all the standard definitions and results for I/O automata (for fairness, composition, etc.) immediately carry over to superposed automata. Furthermore, we show that any fair execution of a superposed automaton, when projected on the underlying module, is a fair execution of the underlying module. In addition, if no output actions of the higher level module are input actions of the underlying module, then every execution of the underlying module is a projection of some execution of the superposed automaton. These results correspond to the notion from [14] that superposition preserves all properties of the underlying algorithm. In addition, we show that when an automaton A is superposed on some other automaton, then the set of schedules of the resulting automaton, when projected on the signature of A, is a subset of the schedules of A alone. Finally, we present a mechanism, analogous to schedule modules for ordinary I/O automata, that allows one to formally specify a problem to be solved by a

higher level module in terms of the state of the lower level module. An example illustrating these extensions is presented in Section 8.2.

Throughout this chapter, we refer to the state of an automaton as being divided into sets of variables, where each set of variables takes on values from a particular domain. For example, we may say that the state of automaton A is divided into two sets of variables X and Y with domains $dom(X)$ and $dom(Y)$, respectively. In this case, we use an ordered pair (x, y) to name a particular state of A, where $x \in dom(X)$ and $y \in dom(Y)$, and we take the set of possible states of A to be the cartesian product $dom(X) \times dom(Y)$. If s is a particular state of A, we let $s|X$ denote the values of the variables of X in state s.

All extensions defined in the section are simply additions to the I/O automaton model. We do not redefine any concepts of the original model, so all of its properties carry over to the extended model.

8.1.1 Unconstrained Automata

When we superpose one module on another, we would like the higher level module not to interfere with the lower level one. In particular, we do not want the higher level module to place constraints on how the lower level module may modify its own variables. Therefore, we will define superposition to apply only when the higher level module is "unconstrained" for the variables of the lower level module. We first define formally what it means for an automaton to be unconstrained for a set of variables. Let X be a set of variables with domain $dom(X)$. An *unconstrained automaton* A for X is an I/O automaton such that there exists a set P of variables with a set of possible initial values $init(P)$ such that:

- $states(A) = dom(P) \times dom(X)$,
- $start(A) = init(P) \times dom(X)$, and
- for every step $((p', x'), \pi, (p, x))$ in $steps(A)$, for all $\hat{x} \in dom(X)$, $((p', x'), \pi, (p, \hat{x}))$ is in $steps(A)$.

Informally, the extra condition on the transition relation says that automaton A places no restrictions on the values of the variables in X following any action. Note, however, that the

set of locally controlled actions enabled in a given state of A may depend on the values of X variables in that state.

Since an unconstrained automaton in an I/O automaton, all the standard I/O automaton definitions for executions, schedules, behaviors, and composition carry over to unconstrained automata. One may think of an "ordinary" I/O automaton as an unconstrained automaton for $X = \emptyset$.

One way to model a layered multicomponent system is to individually superpose pairs of automata and then compose. An equally valid method is to create two entire system layers through composition, and then superpose. In using the latter method, we would like the composition of an unconstrained automaton for X and an unconstrained automaton for Y, with $X \cap Y = \emptyset$, to be an unconstrained automaton for $X \cup Y$. However, this is not the case. Even if the components of the higher layer are each appropriately unconstrained, their composition is not.[1] Therefore, we define a relaxation operator $\mathcal{U}$ that builds an unconstrained automaton from an ordinary one. Let A be an I/O automaton whose state is divided into two sets of variables P and X with domains $dom(P)$ and $dom(X)$ respectively. We define the *relaxation of A with respect to X*, denoted $\mathcal{U}(A, X)$, to be the automaton B as follows:

- $sig(B) = sig(A)$,
- $states(B) = states(A)$,
- $start(B) = \{(p, \hat{x}) : \hat{x} \in dom(X) \wedge \exists x \in dom(X), (p, x) \in start(A)\}$,
- $steps(B) = \{((p', x'), \pi, (p, \hat{x})) : \hat{x} \in dom(X) \wedge \exists x \in dom(X), ((p', x'), \pi, (p, x)) \in steps(A)\}$, and
- $part(B) = part(A)$.

The relaxation operator $\mathcal{U}$ simply constructs the new automaton by adding enough start states and steps to make it unconstrained for X. The following lemma follows immediately from the definitions.

[1] For example, suppose A_X is an unconstrained automaton for X and A_Y is an unconstrained automaton for Y. In the composition of A_X and A_Y, the values of the variables of X are changed only in steps involving actions of A_X. Therefore, any action of A_Y that is not an action of A_X is constrained to leave the values of the variables in X unchanged. Thus, the composition of A_X and A_Y is not unconstrained for $X \cup Y$.

Lemma 8.1: Let A be an I/O automaton whose state is divided into two sets of variables, P and X. Then $\mathcal{U}(A, X)$ is an unconstrained automaton for X.

The following result allows us to prove properties of the schedules of individual unconstrained automata with the knowledge that these properties will carry over to all schedules of the relaxation of the composition.

Lemma 8.2: Let $\{X_i\}_{i \in \mathcal{I}}$ be a set of disjoint sets of variables, and let $\{A_i\}_{i \in \mathcal{I}}$ be a collection of strongly compatible automata, where each A_i is unconstrained for X_i. Let A be the composition $\Pi_{i \in \mathcal{I}} A_i$, and let A_u be the automaton $\mathcal{U}(A, \bigcup_{i \in \mathcal{I}} X_i)$. Then $scheds(A_u) = scheds(A)$ and $fairscheds(A_u) = fairscheds(A)$.

Proof: We know that $scheds(A) \subseteq scheds(A_u)$, since $start(A) \subseteq start(A_u)$ and $steps(A) \subseteq steps(A_u)$ by definition. We show that $scheds(A_u) \subseteq scheds(A)$ using the following construction. Let α_u be an execution of A_u, and let α be identical to α_u, except that $\forall i \in \mathcal{I}, \forall n > 0$, if s_u is the n^{th} state of α_u and s is the n^{th} state of α, then $s|X_i = s'_u|X_i$, where s'_u is the state of α_u immediately preceding the first action of A_i following s_u (if no action of A_i follows s_u, then s'_u is the state immediately after the the last action of A_i in α_u; if no action of A_i occurs in α_u, then s'_u is the initial state of α_u). Note that the value of $s|X_i$ is identical for all states s between to successive actions of A_i in α, and is equal to the value of X_i just before the next step of A_i in α_u.

Clearly $sched(\alpha) = sched(\alpha_u)$. To show that α is a schedule of A, we must show that (1) if s_0 is the first state of α, then $s_0 \in start(A)$, and (2) every step (s', π, s) in α is in $steps(A)$. For condition (1), since each component A_i is unconstrained for X_i, we know that the initial value for x_i may be any value in $dom(X_i)$. Therefore, $s_0 \in start(A)$. For condition (2), we note that if (s', π, s) is the n^{th} step of α, we know from the construction that if $\pi \in acts(A_i)$, then $s'|A_i = s'_u|A_i$, where s'_u is the n^{th} state of α_u, and that π is enabled in state s'_u. Therefore, π is enabled in state s'. And since A_i is unconstrained for X_i, any value is possible for X_i in the resulting state. Furthermore, we know from the construction that if $\pi \notin acts(A_i)$, then $s|A_i = s'|A_i$. Therefore, (s', π, s) is a step of A.

The fairness result follows from the above arguments and the fact that $part(A_u) = part(A)$ by definition of the relaxation operator. ∎

8.1.2 Superposition

In this section, we define the conditions under which one module may be superposed on another, and then define the superposition operator itself.

Requirements for Superposition: In order to provide a sensible semantics for the superposition operator, we define the superposition of one automaton on another only when the two automata satisfy certain compatibility conditions, defined as follows. Let X be a set of variables with domain $dom(X)$. We say that automaton A is *superposable* on automaton B with respect to X iff

1. A is unconstrained for X,
2. $states(B) = dom(X)$, and
3. $sig(A)$ and $sig(B)$ are strongly compatible.

Loosely speaking, the first condition ensures that module B may freely modify its own variables in the superposition. The second condition says that the set of states of the underlying automaton must match the domain for the set of variables on which A is unconstrained. The third condition is the usual restriction for composition of modules.

Superposition Operator: We would like superposition to capture the idea that the higher level automaton is allowed to observe (but not modify) the state of the lower level automaton, and that the lower level automaton is unaware of the variables of the higher level automaton. We want the actions of the superposed automaton to include the actions of both the high level and low level automata, and we wish to allow the possibility of actions that are shared by both automata. This motivates the following definition.

Let X be a set of variables with domain $dom(X)$, and let A and B be automata such that A is superposable on B with respect to X. We define the *superposition of A on B with respect to X*, denoted $C = \mathcal{S}(A, B, X)$, as follows:

- $sig(C) = sig(A) \times^2 sig(\mathrm{B})$,

[2] Usual signature composition.

- $states(C) = states(A)$,
- $start(C) = \{(p, x) \in start(A) : x \in start(B)\}$,
- $steps(C)$ = all steps $((p', x'), \pi, (p, x))$ such that the following conditions hold:
  1. $\pi \in sig(C)$
  2. if $\pi \in sig(A)$, then $((p', x'), \pi, (p, x)) \in steps(A)$
  3. if $\pi \in sig(B)$, then $(x', \pi, x) \in steps(B)$
  4. if $\pi \notin sig(A)$, then $p = p'$
  5. if $\pi \notin sig(B)$, then $x = x'$, and
- $part(C) = part(A) \cup part(B)$.

Informally, the signature of the superposed automaton C is the composition of the signatures of A and B. The states of C are the same as the states of A, and the set of start states of C is the set of all start states of A such that the values of X agree with some start state of B. The most interesting part of the superposition definition is the construction of the set of steps. It says that any step of C for an action of A must also be a step of A. Similarly, any step of C for an action of B must be a step of B, when projected on the variables in X. Essentially, the actions of A and B are enabled just as before, with automaton B placing constraints on the values of the variables in X. The last two conditions of the $steps(C)$ construction simply prevent steps involving only B from modifying the private variables of A, and steps involving only A from modifying the variables in X. That is, if a step of C does *not* involve an action of A, then the private state variables of A must not be modified by the step. Similarly, if a step of C does *not* involve an action of B, then the values of the variables in X are unchanged by the step.

In a step for an action shared by A and B, the private state of A is modified according to the transition relation of A, while the state of X is modified according to the transition relation of B. This should agree with one's intuition about the semantics for such shared actions.

The following lemma states that a superposition of one I/O automaton on another results in a new I/O automaton. This implies that all the standard definitions and results for I/O automata, notably for composition and fairness, immediately carry over to superposed automata.

Lemma 8.3: Let X be a set of variables with domain $dom(X)$, and let A and B be automata such that A is superposable on B with respect to X. Then $C = \mathcal{S}(A, B, X)$ is an I/O automaton.

Proof: We must show that inputs of C are always enabled. That is, we must show that for all states $s' \in states(C)$ and for all actions $\pi \in in(C)$, there exists a state $s \in states(C)$ such that $(s, \pi, s) \in steps(C)$. Let $s' = (p', x')$. There are three cases for $\pi \in in(C)$. For each case, we exhibit an appropriate new state s:

1. $\pi \in sig(A)$ and $\pi \notin sig(B)$. Since A is an unconstrained automaton, we know that $\exists p \in private(A)$ such that $\forall \hat{x} \in dom(X)$, $((p', x'), \pi, (p, \hat{x})) \in steps(A)$. Specifically, if we let $\hat{x} = x'$, then we are done.

2. $\pi \notin sig(A)$ and $\pi \in sig(B)$. Since B is an I/O automaton, we know that $\exists x \in states(B)$ such that $(x', \pi, x) \in steps(B)$. Therefore, since $\pi \notin sig(A)$, $((p', x'), \pi, (p', x)) \in steps(C)$.

3. $\pi \in sig(A)$ and $\pi \in sig(B)$. Since A is an unconstrained automaton, we know that $\exists p \in private(A)$ such that $\forall \hat{x} \in dom(X)$, $((p', x'), \pi, (p, \hat{x})) \in steps(A)$. Furthermore, since B is an I/O automaton, we know that $\exists x \in states(B)$ such that $(x', \pi, x) \in steps(B)$. Therefore, letting $\hat{x} = x$ completes the proof.

In each case, π is enabled from s'.

■

The following two results formalize the notion that properties of the underlying algorithm are preserved in the superposition.

Lemma 8.4: Let X be a set of variables with domain $dom(X)$. Let A and B be automata such that A is superposable on B with respect to X, and let $C = \mathcal{S}(A, B, X)$. Then $execs(C)|B \subseteq execs(B)$ and $fairexecs(C)|B \subseteq fairexecs(B)$.

Proof: Let α be a (fair) execution of C. By definition of superposition, if (s', π, s) is a step of α and $\pi \notin acts(B)$ then $s|X = s'|X$. Therefore, $\alpha|B$ is a (fair) execution of B. (The fairness result follows from the fact that $part(B) \subseteq part(C)$, so any execution fair to the classes of C must also be fair to the classes of B.)

■

In general, it is not the case that every execution of the lower level automaton is a projection of an execution of the superposed automaton. For example, lower level automaton B may have

π as an input action, so its set of executions include executions in which π occurs multiple times. If automaton A is defined to have π as an output action such that π occurs at most once in every execution of A, then none of the executions of B in which π occurs more than once are projections of executions of the superposition of A on B. However, when no output actions of the higher level automaton are inputs to the lower level automaton, the converse of Lemma 8.4 holds, and we have the following result.

Lemma 8.5: Let X be a set of variables with domain $dom(X)$. Let A and B be automata such that A is superposable on B with respect to X, and let $C = S(A, B, X)$. If $in(B) \cap out(A) = \emptyset$, then $execs(C)|B = execs(B)$ and $fairexecs(C)|B = fairexecs(B)$.

Proof: From Lemma 8.4, we know that $execs(C)|B \subseteq execs(B)$ and $fairexecs(C)|B \subseteq fairexecs(B)$. Let β be a (fair) execution of B. Since $in(B) \cap outA = \emptyset$, we know that the higher level component A has no control over which actions of $acts(B)$ occur in an execution of C. Furthermore, only the actions of B may change the variables in X in the superposition. Therefore, since a locally controlled action of B is enabled from state s in C iff it is enabled from state $s|X$ in B, there must exist some (fair) execution γ of C such that $\gamma|B = \beta$. Thus, $execs(B) \subseteq execs(C)|B$ and $fairexecs(B) \subseteq fairexecs(C)|B$. ∎

The next result says that when an automaton A is superposed on some other automaton, then the set of schedules of the resulting automaton, when projected on the signature of A, is a subset of the schedules of A alone. This is very important because it allows us to prove safety properties about A alone with the knowledge that these properties will hold when A is superposed on some other automaton.

Lemma 8.6: Let X be a set of variables with domain $dom(X)$. Let A and B be automata such that A is superposable on B with respect to X, and let $C = S(A, B, X)$. Then $scheds(C)|sig(A) \subseteq scheds(A)$.

Proof: Let γ be an execution of C. We construct α from γ by the following steps:

1. Remove from γ all actions not in $sig(A)$. This may create sequences of states not separated by actions.

2. Replace the sequence of states between each pair of successive actions by the last state in that sequence.

Clearly, α is an alternating sequence of states of A and actions of A. To show that $\alpha \in execs(A)$, we must show that the first state of α is in $start(A)$ and that if $\sigma = ((p', x'), \pi, (p, x))$ is a step in α, then $\sigma \in steps(A)$. For both of these, we use the following fact.

> *Fact:* If a sequence of states from γ is replaced in step 2 of the construction by the single state (p, x), then every state in that sequence has p as the value of the private state of A.
>
> *Proof of Fact:* In γ, each of these states is separated by an action not in the signature of A. From the definition of superposition, we know that any step in γ not involving an action of A does not change the values of the private variables of A. Since (p, x) is the last state in the sequence, every state in the sequence must have p as its first component.

From the above fact, we know that the first state of γ and the first state of α agree on the values of the private variables of A. Since A is unconstrained, any value in $dom(X)$ is an allowable value for the second component of the start state. Therefore, the first state of α is a start state of A.

If $\sigma = ((p', x'), \pi, (p, x))$ is a step in γ, then we know that π occurs from state (p', x') in α. Furthermore, from the above fact, we know that π results in state $(p, \hat{x})$ in γ for some $\hat{x} \in dom(X)$. So, we know that $steps(A)$ contains the step $((p', x'), \pi, (p, \hat{x}))$ for some $\hat{x} \in dom(X)$. Therefore, since A is unconstrained for X, we know that $steps(A)$ contains the step $((p', x'), \pi, (p, \hat{x}))$ for *all* $\hat{x} \in dom(X)$, and specifically for $\hat{x} = x$. This completes the proof. ■

Note that not all schedules of A are necessarily possible in the superposition, since certain states reachable in A alone may not be reachable in the superposition. For example, suppose a particular action π of A is enabled only when a variable $x \in X$ has a particular value v, and suppose that the automaton B on which A is superposed is defined to never set $x = v$. Since A alone may set x to any value (by the definition of unconstrained for X), the action π may occur in behaviors of A. However, by the definition of superposition, there is no step of $\mathcal{S}(A, B, X)$ that results in $x = v$, so π is never enabled. This is a perfectly natural and desirable property of superposition, for it says that the state of the lower layer affects the behavior of the higher layer.

Another interesting fact is that $\mathit{fairscheds}(C)|\mathit{sig}(A)$ and $\mathit{fairscheds}(A)$ are incomparable. We know from the above paragraph that $\mathit{fairscheds}(A) \not\subseteq \mathit{fairscheds}(C)|\mathit{sig}(A)$. But it is also the case that $\mathit{fairscheds}(C)|\mathit{sig}(A) \not\subseteq \mathit{fairscheds}(A)$, as witnessed by the following example. Suppose that A has only two actions, π_1 and π_2, each in its own class of the partition. Furthermore, suppose that both events are enabled exactly when $x = 0$. Now, suppose that B has exactly one action π_3, that toggles the value of x between 0 and 1, and is always enabled. In the superposition of A on B, a fair schedule would be $\pi_1, \pi_3, \pi_3, \pi_1, \pi_3, \pi_3, \pi_1, \ldots$, in which the class containing π_2 is given a chance to take a step only when $x = 1$. However, the infinite schedule $\pi_1, \pi_1, \pi_1, \ldots$ is *not* a fair schedule of A alone: Since the schedule consists of infinitely many π_1 actions, it must be that π_1 is enabled from every state of the corresponding execution. Therefore, $x = 0$ in all states of that execution. But π_2 is also enabled whenever $x = 0$, yet the class containing π_2 is never given a chance to take a step, so the schedule is not fair.

The reason for the above fact is that the preconditions for the locally controlled actions of A are allowed to depend upon the values of the variables in X. Keeping this in mind, consider the following additional condition on unconstrained automata. If A is an unconstrained automaton for X, then A is said to be *completely unconstrained for* X iff for for all actions $\pi \in \mathit{sig}(A)$, if $((p', x'), \pi, (p, x)) \in \mathit{steps}(A)$ then for all $\hat{x} \in \mathit{dom}(X)$, there exists a state $(\hat{p}, x) \in \mathit{states}(A)$ such that $((p', \hat{x}), \pi, (\hat{p}, x)) \in \mathit{steps}(A)$. In other words, whether or not an action of A is enabled can depend only upon the values of its private variables. The only way for A to make any use of the variables of X would be for it to modify its own local variables according to what it observes in X, causing other actions of A to become enabled or disabled. Modifying the definition of superposable to require the higher level automaton to be completely unconstrained for X would allow us to prove that $\mathit{fairscheds}(C)|\mathit{sig}(A) \subseteq \mathit{fairscheds}(A)$, but would result in a significant loss of expressive power. So, rather than require this condition outright, we state the following lemma, which says that *if* an automaton happens to be completely unconstrained, then the containment result holds for its fair schedules. This gives us more flexibility in the use of the model.

Lemma 8.7: Let X be a set of variables with domain $\mathit{dom}(X)$. Let A and B be automata such that A is completely unconstrained for X and A is superposable on B with respect to X. Let $C = \mathcal{S}(A, B, X)$. Then $\mathit{fairscheds}(C)|\mathit{sig}(A) \subseteq \mathit{fairscheds}(A)$.

Proof: Analogous to that of Lemma 8.6, but noting that the actions of A are enabled independently of the value of X and applying the definition of fairness. ■

The definition of an unconstrained automaton A for X requires that the value of X may be changed arbitrarily with each step of A. However, a more natural way to describe the behaviors of a module to be superposed on another module might be to allow the values of X to change *between* the steps of A as well. For this, we define the notion of an "extended execution" in which several states may occur between two successive actions. If A is an unconstrained automaton for X, we define an *extended execution* of A to be a sequence α of states in $states(A)$ and actions in $acts(A)$, beginning with a state in $start(A)$, such that:

1. if a state-action-state sequence $s'\pi s$ appears in α, then (s', π, s) is in $steps(A)$,
2. if two states s' and s appear consecutively in α, then they differ only in the value of X, and
3. no two actions appear consecutively in α.

We define fairness for extended executions exactly as for ordinary executions. We let $extexecs(A)$ and $fairextexecs(A)$ denote the sets of extended executions and fair extended executions of A, respectively. If α is a sequence of states and actions and Π is a set of actions, we define the notation $\alpha \| \Pi$ to be the sequence that results from deleting from α exactly those actions not in Π. Using extended executions instead of ordinary executions, we get the desired fairness result:

Lemma 8.8: Let X be a set of variables with domain $dom(X)$. Let A and B be automata such that A is superposable on B with respect to X, and let $C = \mathcal{S}(A, B, X)$. Then $execs(C) \| sig(A) \subseteq extexecs(A)$ and $fairexecs(C) \| sig(A) \subseteq fairextexecs(A)$.

Proof: Let α be an execution of C. From the definition of superposition, we know that α begins with a state in $start(A)$, and that any step (s', π, s) occurring in α with $\pi \in sig(A)$ must be a step of A. Also by the definition of superposition, for any step (s', π, s) where π is *not* in $sig(A)$, s' and s must differ only in the value of X. Therefore, $\alpha \| sig(A)$ is an extended execution of A by definition. If α is fair, then since α and $\alpha | sig(A)$ contain the same sequence of states, and since $part(A) \subseteq part(C)$, we know that $\alpha \| sig(A)$ is a fair extended execution of A. ■

8.1.3 Partial Execution Modules

It is important to have a formal mechanism for specifying the problem to be solved by an automaton. Schedule modules, as described in Section 2.4, permit us to specify the allowable schedules of a module in terms of the actions that occur the boundary with its environment. However, if an automaton A is to be superposed on top of some underlying automaton B, then we would like to specify the allowable behaviors of A not only in terms of the actions that occur at its external interface, but also in terms of the internal state of B. To accomplish this, we define a new specification mechanism called a "partial execution module."

Let X be a set of variables with domain $dom(X)$, and let Π be a set of actions. A *partial execution* for Π and X is defined to be a sequence of states and actions, beginning with a state, such that each state is in $dom(X)$, each action is in Π, and each action is immediately followed by a state. Note that a partial execution may contain several states between two consecutive actions. A *partial execution module* H consists of

- $sig(H)$, an external action signature,
- $vars(H)$, a set of variables with domain $dom(vars(H))$, and
- $pexecs(H)$, a set of partial executions for $sig(H)$ and $vars(H)$.

A partial execution module H defines a problem to be solved by an unconstrained automaton for $vars(H)$ with external signature $sig(H)$. In order to define what it means for an automaton to "solve" H, we need a way to extract partial executions from extended executions: Let X be a set of variables with domain $dom(X)$, let Π be a set of actions, and let α be an extended execution of any automaton that is unconstrained for X. We define $\alpha|(\Pi, X)$, the *partial execution for* Π *and* X *in* α, to be the same as α, except that each state s is replaced by its projection on X and each action not in Π is deleted. If A is an unconstrained automaton for X with external signature Π, we define $pexecs(A, X)$ to be the set $\{\alpha|(\Pi, X) : \alpha \in fairextexecs(A)\}$.

An automaton A is said to *solve* a partial execution module H iff $pexecs(A, vars(H)) \subseteq pexecs(H)$.

8.1.4 Superposition for Partial Executions

Lynch and Tuttle define composition for both automata and schedule modules. So far, we have defined the superposition of one automaton on another, but have not yet defined an analogous operator for superposing a partial execution module on another module. We now complete the theory by defining the superposition of a set of partial executions on a set of ordinary executions.

Let X be a set of variables and let Π and Δ be sets of actions. Let ∂ be a set of partial executions for Π and X, and let Φ be a set of alternating sequences of states in of $dom(X)$ and actions in Δ. Let $\mho$ be the set of all alternating sequences of states of X and actions of $\Pi \cup \Delta$. We now define the *superposition of ∂ on Φ with respect to X*. Overloading the $\mathcal{S}$ notation, we define

$$\mathcal{S}(\partial, \Phi, X) = \{\alpha \in \mho : \alpha||\Pi \in \partial \wedge \alpha|\Delta \in \Phi\}.$$

In other words, for each element α of $\mathcal{S}(\partial, \Phi, X)$, deleting all actions from α except those in Π results in a partial execution in ∂, and projecting α on the actions of Δ results in an execution in Φ.

The following result says that the set of fair behaviors of a superposition of A on B with respect to X is the same as the set of behaviors resulting from the superposition of $pexecs(A, X)$ on the fair executions of B.

Lemma 8.9: Let X be a set of variables. If automaton A is superposable on automaton B with respect to X, then $fairbehs(\mathcal{S}(A, B, X)|(\Pi, X)) = behs(\mathcal{S}(pexecs(A), fairexecs(B), X))$.

Proof: If β is a fair behavior of $\mathcal{S}(A, B, X)$, let α be the corresponding execution. By Lemma 8.8, $\alpha||sig(A)$ is a fair extended execution of A, so $\alpha|(sig(A), X) \in pexecs(A)$. And by Lemma 8.4, $\alpha|B \in fairexecs(B)$, so $fairbehs(\mathcal{S}(A, B, X)|(\Pi, X)) \subseteq behs(\mathcal{S}(pexecs(A), fairexecs(B), X))$.

To show the other direction, let α be an element of $\mathcal{S}(pexecs(A), fairexecs(B), X)$. We know from the definition of superposition of partial executions, we know that $\alpha||\Pi \in pexecs(A)$. Therefore, there exists a fair extended execution α' of A such that $\alpha = \alpha'|(ext(A), X)$. Also from the definition of superposition of partial executions, we know that $\alpha|\Delta \in fairexecs(B)$. Since the states of α an α' are identical with respect to X, we know that for each step (s', π, s)

of $\alpha|\Delta$, there exists in α' a pair of consecutive states $\hat{s'}$ and $\hat{s}$ such that $\hat{s'}|X = s'$ and $\hat{s}|X = s$. We construct α'' by inserting each action of $\alpha|\Delta$ between the corresponding pair of states in α'. To complete the proof, we must show that α'' is a fair execution of $\mathcal{S}(A, B, X)$. We know that α'' begins with an initial state of A. Now, we consider the four possible cases for each step (s', π, s) in α'':

1. If $\pi \in sig(A)$, then $(s', \pi, s) \in steps(A)$, since α' is an extended execution of A.

2. If $\pi \in sig(B)$, then $(s'|X, \pi, s|X) \in steps(B)$, because of our construction of α'' from α and the fact that $\alpha|\Delta$ is an execution of B.

3. If $\pi \notin sig(A)$, then s' and s differ only in the value of X by definition of an extended execution.

4. If $\pi \notin sig(B)$, then $s'|X = s|X$, again because $\alpha|\Delta$ is an execution of B.

Therefore, α'' is an execution of $\mathcal{S}(A, B, X)$. To show that α'' is fair, we note that $part(\mathcal{S}(A, B, X)) = part(A) \cup part(B)$ by the definition of superposition, and we consider the classes of A and B separately. We know that α' is a fair extended execution of A. Therefore, since $\alpha''||A = \alpha'$, α'' is fair to the classes of A. Similarly, since $\alpha|\Delta$ is a fair execution of B, and $\alpha''|B = \alpha|\Delta$, we know that α is fair to the classes of B. ∎

8.2 Example: Global Snapshot

In this section, we illustrate the superposition model extensions with the Chandy-Lamport global snapshot algorithm [13]. We begin by defining the global snapshot problem with a partial execution module G. Then, we describe the global snapshot algorithm as an automaton to be superposed on an application program. Finally, we give a complete proof that the global snapshot algorithm solves partial execution module G.

8.2.1 Problem Specification

We consider systems of processes that communicate by sending messages over a network. The network guarantees eventual one-time delivery of each message such that messages sent from

a given process to each other process arrive in the order sent (pairwise FIFO). The goal of a *distributed global snapshot* protocol is to produce a global state of a system (states of all processes and the set of messages in transit) during an ongoing computation. The snapshot algorithm is not allowed to interfere with the computation of the rest of the system. For example, the snapshot algorithm cannot halt the system. In addition, the snapshot obtained must be both *consistent* and *recent*. By consistent, we mean that the snapshot is a state that could have occurred in some execution[3] of the underlying system. By recent, we mean that there exists some execution of the underlying system that contains the following states in order (with possibly other states in between):

1. the *initiation* state, the state of the system when the snapshot protocol is initiated,

2. the global state of the system reported by the snapshot algorithm,

3. the *termination* state, the state of the system when the snapshot algorithm terminates.

In other words, the snapshot state is a state that "could" have occurred between the initiation and termination states of the system.

The global snapshot problem, and protocols for solving it, are neatly specified using the superposition definitions defined in the previous section. We view the snapshot algorithm as a layer to be superposed on top of the application layer. We begin by specifying the signature for each of the underlying application processes and a schedule module for the network. Then, we present a partial execution module G that formalizes the problem statement given above.

Application Processes

Let $\mathcal{I}$ be a finite set of names for the communicating processes in the application program. Let $\mathcal{M}$ be a universal set of messages containing a special marker symbol (#). For each $i \in I$, we fix a corresponding application process u_i. Each process u_i is modelled as an automaton having a set of state variables X_i with domain $states(u_i)$, and the following signature:

[3]The states of these executions include not only the states of the application processes, but also the set of messages in transit in the system (i.e., the state of the network).

Input actions: $\text{RCV}(m, j, i), m \in \mathcal{M}, j \in \mathcal{I}$
Output actions: $\text{SEND}(m, i, j), m \in \mathcal{M}, j \in \mathcal{I}$

The input actions represent u_i receiving a message m from u_j, and the output actions represent u_i sending a message m to u_j. Associated with u_i are two sets, *out-chans(i)* and *in-chans(i)*, both subsets of $\mathcal{I}$. One may think of *out-chans(i)* as identifying those application processes to whom u_i may send messages, the "outgoing channels" of u_i. Similarly, *in-chans(i)* identifies the application process from whom u_i may receive messages, the "incoming channels" of u_i. Let graph CHANS(V, E) be the graph with $|\mathcal{I}|$ vertices uniquely labeled by the elements of $\mathcal{I}$ and $E = \{(i \in \mathcal{I}, j \in \mathcal{I}) : j \in \textit{out-chans(i)}\}$. We allow the set *out-chans(i)*, for each $i \in \mathcal{I}$, to be arbitrary, except that the graph CHANS must be strongly connected. For each $i \in \mathcal{I}$, the set *in-chans(i)* is defined, as one would expect, to be the set of all $j \in \mathcal{I}$ such that $i \in$ *out-chans(j)*.

A sequence β of actions of u_i is said to be *well-formed* for i iff for all $m \in \mathcal{M}$, and for all $j, k \in \mathcal{I}$, if $\text{SEND}(m, i, j)$ occurs in β, then $j \in$ *out-chans(j)* and $m \neq \#$, and if $\text{RCV}(m, k, i)$ occurs in β, then $k \in$ *in-chans(k)*. (The technical restriction on the symbol # is present because # is a special message in the snapshot algorithm, and therefore must be distinguished from the ordinary messages of the application processes.)

We require that u_i preserve well-formedness for i, but make no other restrictions on the allowable behaviors of u_i. We make no restrictions on the domain *states*(u_i).

Correspondence Relations

In specifying the network, as well as the global snapshot problem, we use a correspondence relation technique similar to that of [18]. Let t denote a text string. We define a *message action for* t to be any action of the form $t(m, i, j)$, where $m \in \mathcal{M}$ and $i, j \in \mathcal{I}$. Let β be a sequence of actions, and let x and y be text strings. Let Π_x be the set of events[4] for message actions for x in β, and let Π_y be the set of events for message actions for y in β. Let $\mathcal{C}$ be a binary relation on the set of events in an execution onto itself, and let us say that two events

[4] We uses the term *event* here to refer to a particular occurrence of an action in the sequence.

in an execution *correspond* iff the relation is true for that pair of events. We say that C is a *correspondence relation* for x and y in β iff the first four of the following conditions hold, and is a *live correspondence relation* iff all of the following conditions hold.

1. Corresponding events have identical arguments.
2. Each event $\pi_y \in \Pi_y$ corresponds to exactly one event $\pi_x \in \Pi_x$, and π_x precedes π_y in β.
3. Each event $\pi_x \in \Pi_x$ corresponds to at most one event in Π_y.
4. If $x(m,i,j)$ precedes $x(m',i,j)$ in β, and $y(m,i,j)$ and $y(m',i,j)$ are their corresponding events, then $y(m,i,j)$ precedes $y(m',i,j)$ in β.
5. For each event $\pi_x \in \Pi_x$, there exists a corresponding event in Π_y.

An intuitive explanation of these conditions follows their use in specifying the network. The following lemma states a transitivity property of correspondence relations.

Lemma 8.10: Let β be a sequence of actions, and let x, y and z be text strings. If C_{xy} is a (live) correspondence relation for x and y in β, and C_{yz} is a (live) correspondence relation for y and z in β, then there exists a (live) correspondence relation C_{xz} for x and z in β.

Proof: Let C_{xz} be defined as follows. Two events π_x and π_z in β correspond iff there exists an event π_y in β such that π_x corresponds to π_y according to C_{xy} and π_y corresponds to π_z according to C_{yz}. The properties of a (live) correspondence relation follow immediately. ■

The Network

Rather than modelling the network as an explicit I/O automaton, we define an action signature for the network and then define a well-formedness property of sequences of those actions that characterizes the desired behaviors of the network. The signature of the network is as follows:

Input actions: $\text{SEND}(m,i,j)$, $m \in \mathcal{M}, i,j \in \mathcal{I}$

Output actions: $\text{RCV}(m,i,j)$, $m \in \mathcal{M}, i,j \in \mathcal{I}$

If β is a sequence of actions, then β is *network admissible* iff there exists a live correspondence relation for SEND and RCV in β. This means that (1) a SEND and RCV correspond only if

they match on the arguments m, i, and j, (2) each RCV corresponds to exactly one SEND, and the SEND occurs earlier, (3) for each SEND event, there corresponds at most one RCV event, (4) messages between pairs of processes are delivered in the order sent, and (5) each message sent is eventually received. The fifth condition is the *liveness* property we assume to be guaranteed by the network.

The Application System

In defining our correctness condition for the global snapshot algorithm, we would like to express the notion that the application processes should not be able to tell whether they are running in a system with the snapshot protocol, or in a system without the snapshot protocol. Therefore, we explicitly define the set of allowable behaviors of the "system without the snapshot protocol" to form the basis of our correctness condition.

Let $U = \Pi_{i \in \mathcal{I}} u_i$, the composition of all the application processes. We define the set of fair behaviors of the application system, denoted $fairbehs(S_{app})$ to be the set of behaviors of all network admissible fair executions of U. We will state the correctness conditions for a global snapshot protocol in terms of $fairbehs(S_{app})$. In doing so, it will be helpful to have the following definitions. Let $seqs(\mathcal{M})$ be the set of all sequences of elements of $\mathcal{M}$, including the empty sequence ϵ. Let α be element of $fairbehs(S_{app})$, and let $\mathcal{C}$ be a correspondence relation for SEND and RCV in α. (We know there is one, by the definition of network admissible.) If α' is a prefix of α, we define $in\text{-}transit_{\alpha',\mathcal{C}} : (\mathcal{I} \times \mathcal{I}) \rightarrow seqs(\mathcal{M})$ as follows. For each pair $i, j \in \mathcal{I}$, $in\text{-}transit_{\alpha',\mathcal{C}}(i,j)$ is the sequence of messages m such that SEND(m,j,i) occurs in α', but the corresponding RCV(m,j,i) does not, ordered according to the SEND events. In other words, for each i,j pair, we have the sequence of messages sent from u_j to u_i, but not yet delivered to u_i. We define *in-transit* analogously for the schedule of α, and for executions or partial executions whose schedules, projected on U, are in $fairbehs(S_{app})$.

Partial Execution Module G

In this section, we specify the global snapshot problem by defining a partial execution module G. Let *chans* name the set of all possible functions from elements of $\mathcal{I}$ to elements of $seqs(\mathcal{M})$, and let *all-chans* name the set of all possible functions from elements of $\mathcal{I}$ to elements of *chans*.

The signature $sig(G)$ is as follows:

Input actions:	START_i, $i \in \mathcal{I}$
	$\text{SEND}(m,i,j)$, $m \in \mathcal{M}, i,j \in \mathcal{I}$
	$\text{MSG\_RCV}(m,i,j)$, $m \in \mathcal{M}, i,j \in \mathcal{I}$
Output actions:	$\text{MSG\_SEND}(m,i,j)$, $m \in \mathcal{M}, i,j \in \mathcal{I}$
	$\text{RCV}(m,i,j)$, $m \in \mathcal{M}, i,j \in \mathcal{I}$
	$\text{DONE}_i(a,c)$, $i \in \mathcal{I}$, $a \in states(u_i)$, $c \in chans$

Let α be a sequence of $states(U)$ and actions of G. If a START action (for any i) occurs in α and exactly one DONE_i action occurs in α for each $i \in \mathcal{I}$, then we let $\alpha = \alpha_1\alpha_2\alpha_3$, where α_2 begins with the first START action and ends with the last DONE action. Furthermore, we define $snap(\alpha)$ to be the pair $(s \in states(U), c \in all\text{-}chans)$ such that $\forall i \in \mathcal{I}$, if $\text{DONE}_i(a_i, c_i)$ occurs in α, then $s|u_i = a_i$ and $c(i) = c_i$. In other words, $snap(\alpha)$ is the collective state of the system (including messages in transit) reported in the DONE_i actions for all $i \in \mathcal{I}$.

Since the snapshot algorithm has no control over its input actions, we require that G behave properly only when its environment, namely the network and the application processes, is well-behaved. Let α be a sequence of $states(U)$ and actions of G. We say that α is *admissible* iff (1) there exists a live correspondence relation C_α for MSG_SEND and MSG_RCV in α, and (2) $\forall i \in \mathcal{I}, \alpha|u_i \in execs(u_i)$. We place constraints on the behavior of G only for admissible partial executions.

Let α be a sequence of $states(U)$ and actions of G. Then $\alpha \in pexecs(G)$ iff the following condition holds. If α is admissible, then

1. $sched(\alpha)|U \in fairbehs(S_{app})$
2. if a START action occurs in α, then
   (a) $\forall i \in \mathcal{I}$, exactly one DONE_i occurs in α, and
   (b) $\exists \beta = \beta_1\beta_2 s \beta_3\beta_4$, an execution of U, with correspondence relation C_β for SEND and RCV in β, such that
      i. $\forall i \in \mathcal{I}, \beta|u_i = \alpha|u_i$,

ii. the last state of α_1 is the last state of β_1, and $in\text{-}transit_{\beta_1,\mathcal{C}_\beta} = in\text{-}transit_{\alpha_1,\mathcal{C}_\alpha}$,

iii. $snap(\alpha) = (s,\ in\text{-}transit_{\beta_1\beta_2,\mathcal{C}_\beta})$, and

iv. the first state of α_3 is the first state of β_4, and $in\text{-}transit_{\beta_1\beta_2 s\beta_3,\mathcal{C}_\beta} = in\text{-}transit_{\alpha_1\alpha_2,\mathcal{C}_\alpha}$.

Condition (1) captures the idea that the computation of the application processes in the system with the snapshot algorithm should be a legal computation in the application system alone. Condition (2) concerns the snapshot itself. It says that if a snapshot is requested, then (a) snapshot information eventually is reported for each application process $u_i \in \mathcal{I}$, and (b) the snapshot produced must be a consistent recent state of the system. Note the similarity of part (b) with the informal statement of the global snapshot problem given at the beginning of this section.

In the specification of the global snapshot problem, we used the states of the underlying algorithm to express the recency condition. This would not have been possible with an ordinary schedule module specification, and points out the need for partial execution modules in conjunction with superposition.

8.2.2 The Algorithm

In this section, we use the I/O automaton model with superposition extensions to describe the global snapshot algorithm of Chandy and Lamport [13]. In the original paper by Chandy and Lamport, the snapshot algorithm is described being "superimposed" on top of the application program. However, the algorithm is actually presented as a modification to the underlying application program, and care is taken to ensure that the modification does not disrupt the application. Chandy and Misra [14] describe the algorithm in UNITY, and Nour [51] recasts that work using I/O automata. In both of these presentations, however, the distinctions between the snapshot algorithm and the underlying application become blurred. That is, the UNITY program resulting from superposing the snapshot algorithm on the application is monolithic; it does not preserve the essential separation of actions under the control of the snapshot protocol and the actions under the control of the application. This is partly due to the lack of separation of inputs and outputs, but is largely due to the absence of a mechanism for partitioning the actions of a program into separate processes. Using I/O automaton superposition, we are able

to achieve a formal separation of the application program from the global snapshot protocol. The "built-in" partition of locally controlled actions of an I/O automaton allows us to model the actions of the snapshot protocol and the actions of the application as being under the control of *different* processes.

To model the global snapshot algorithm, we define a *snapshot automaton* p_i for each $i \in I$. In a sense, p_i encapsulates the corresponding application automaton u_i, acting as a buffer between the application processes and the network. Because of this structure, the SEND and the RCV actions of the application are no longer shared with the network, but are instead shared with the snapshot automaton. The snapshot automaton, in turn, interacts with the network using MSG_SEND and MSG_RCV actions to avoid naming conflicts. We postulate a renamed network, where the SEND actions are renamed to be MSG_SEND actions and the RCV actions are renamed to be MSG_RCV actions. In this way, the encapsulation structure is supported without renaming the actions of the application processes.

Each snapshot automaton p_i, $i \in \mathcal{I}$ has several state components. The components *state-snapped* and *chan-snapped*$[j]$, $j \in \mathcal{I}$ are boolean variables, initially false, that record whether or not the state of u_i and the states of the various "incoming channels" adjacent to u_i have been recorded for the snapshot. The components *in-queue*$[j]$ and *out-queue*$[j]$, for each $j \in \mathcal{I}$, are queues of messages, initially empty. These contain messages that are waiting to be delivered to (or sent from) application process u_i. Recall that a message for process u_i from process u_j is not delivered directly to process u_i from the network. Instead, the message is delivered to the snapshot automaton p_i, which places the message in *in-queue*$[j]$ for later delivery. Similarly, when process u_i sends a message to process u_j, the SEND action is not shared with the network automaton. Instead, the snapshot automaton p_i puts the message in *out-queue*$[j]$ and later sends out the message. The component *snapshot*, initially undefined, takes on values in *states*(u_i), and is used to record the snapshot of the application's state at u_i. Similarly, *chan-state*$[j]$, for $j \in \mathcal{I}$ is an initially empty queue of messages that is used to record the state of the incoming channel from process j. Finally, the state component *done* is a boolean variable indicating whether or not p_i has reported the results of its local snapshot in a DONE$_i$ action.

The signature and transition relation are shown in Figure 8.2.2. In the code, if q is a queue, then head(q) refers to the first item in the queue (i.e., the one that would be dequeued next), or

nil if the queue is empty. The function tail(q) refers to the queue that results from dequeuing head(q). If q has zero or one elements, then tail(q) = *nil*. Since the snapshot automaton p_i is designed to be superposed on top of the application process automaton u_i, we use the state component *app* to refer to the state of the application process automaton u_i. One can easily check that p_i is unconstrained (in fact, completely unconstrained) for $\{app\}$. (See Lemma 8.11.) A step (s', π, s) appears in the transition relation for p_i iff the precondition for π holds in state s', and state s is derived from s' according to the assignments in the effect of π. If no precondition is given, then it is assumed to be true in all states.

The partition $part(p_i)$ is defined as follows. For each $j \in \mathcal{I}$, there are two classes: all actions of the form RCV(m, i, j), $m \in \mathcal{M}$, and all actions of the form MSG_SEND(m, i, j), $m \in \mathcal{M}$. Finally the action DONE$_i$ is in a separate class.

The snapshot algorithm is initiated at p_i either by a START$_i$ action from the environment or by receipt of a marker message (#). The environment may generate any number of START messages for any number of snapshot processes. However, we view the first START message in an execution as the start of the global snapshot protocol. The first time a START$_i$ occurs or p_i receives a marker, p_i records the local state of the underlying application automaton u_i (and records the state of the incoming channel on which the marker was received as being empty), places a marker in the queue for all of its outgoing channels, and begins keeping track of all messages received on its incoming channels in the *chan-state* variables. Any later START$_i$ message is ignored. Any later receipt of a marker on a given "channel" j causes p_i to set *chan-snapped*[j] to true, which prevents p_j from adding later messages to *chan-state*[j]. Once the state of u_i and the states of all incoming channels have been snapped, p_i may issue a $DONE$ action, reporting the snapshot information.

We now prove some simple properties of p_i. The compositionality properties of I/O automata will allow us to use these local results in proving properties of larger systems containing p_i.

Lemma 8.11: Automaton p_i is completely unconstrained for *app*.

Proof: By inspection of the code for p_i, *app* never appears in a precondition, and never appears on the left hand side of an assignment in an effect. ∎

Input actions: START_i
$\text{MSG\_RECEIVE}(m, j, i)$, $m \in \mathcal{M}, j \in \mathcal{I}$
$\text{SEND}(m, i, j)$, $m \in \mathcal{M}, j \in \mathcal{I}$

Output actions: $\text{RCV}(m, j, i)$, $m \in \mathcal{M}, j \in \mathcal{I}$
$\text{MSG\_SEND}(m, i, j)$, $m \in \mathcal{M}, j \in \mathcal{I}$
$\text{DONE}_i(a, c)$, $s \in \mathit{states}(u_i), c \in \mathit{chans}$

- START_i
  Effect: if $s'.\mathit{state\text{-}snapped} = \text{false}$ then
  $s.\mathit{snapshot} = s'.\mathit{app}$
  $s.\mathit{state\text{-}snapped} = \text{true}$
  $\forall k \in \mathit{out\text{-}chans}(i)$, $s.\mathit{out\text{-}queue}[k] = \# \circ s'.\mathit{out\text{-}queue}[k]$

- $\text{MSG\_RCV}(m, j, i)$
  Effect: if $m = \#$ then
  if $s'.\mathit{state\text{-}snapped} = \text{false}$ then
  $s.\mathit{snapshot} = s'.\mathit{app}$
  $s.\mathit{state\text{-}snapped} = \text{true}$
  $\forall k \in \mathit{out\text{-}chans}(i)$, $s.\mathit{out\text{-}queue}[k] = \# \circ s'.\mathit{out\text{-}queue}[k]$
  $s.\mathit{chan\text{-}snapped}[j] = \text{true}$
  else
  $s.\mathit{in\text{-}queue}[j] = m \circ s'.\mathit{in\text{-}queue}[j]$
  if $s'.\mathit{state\text{-}snapped} = \text{true} \wedge s'.\mathit{chan\text{-}snapped}[j] = \text{false}$ then
  $s.\mathit{chan\text{-}state}[j] = m \circ s'.\mathit{chan\text{-}state}[j]$

- $\text{SEND}(m, i, j)$
  Effect: $s.\mathit{out\text{-}queue}[j] = m \circ s'.\mathit{out\text{-}queue}[j]$

- $\text{RCV}(m, j, i)$
  Precondition: $m = \text{head}(s'.\mathit{in\text{-}queue}[j])$
  Effect: $s.\mathit{in\text{-}queue}[j] = \text{tail}(s'.\mathit{in\text{-}queue}[j])$

- $\text{MSG\_SEND}(m, i, j)$
  Precondition: $m = s'.\text{head}(\mathit{out\text{-}queue}[j])$
  Effect: $s.\mathit{out\text{-}queue}[j] = \text{tail}(s'.\mathit{out\text{-}queue}[j])$

- $\text{DONE}_i(a, c)$
  Precondition: $s'.\mathit{state\text{-}snapped} = \text{true}$
  $\forall j \in s'.\mathit{in\text{-}chans}$, $s'.\mathit{chan\text{-}snapped}[j] = \text{true}$
  $s'.\mathit{done} = \text{false}$
  $a = s'.\mathit{snapshot}$
  $\forall j \in \mathcal{I}$, $c(j) = s'.\mathit{chan\text{-}state}(j)$
  Effect: $s.\mathit{done} = \text{true}$

Figure 8-1: Global Snapshot Automaton p_i.

Lemma 8.12: Let α be a fair execution of p_i. Then there exists a live correspondence relation for SEND and MSG_SEND in α, and there exists a live correspondence relation for MSG_RCV and RCV in α.

Proof: By definition, a SEND(m,i,j) action for p_i places m into *out-queue*$[j]$. Only a MSG_SEND(m,i,j) action can remove m from that queue, and the only precondition for a MSG_SEND(m,i,j) action is that m is at the head of *out-queue*$[j]$. Therefore, since α is a fair execution, exactly one MSG_SEND(m,i,j) event eventually occurs for each element of *out-queue*$[j]$, in the order of the SEND events. This implies that there is a live correspondence relation for SEND and MSG_SEND in α. Similarly, by definition a MSG_RCV(m,j,i) action places m into *in-queue*$[j]$. Since α is a fair execution, exactly one RCV(m,j,i) event eventually occurs for each element of *in-queue*$[j]$, in the order of the MSG_RCV events. Therefore, there is a live correspondence relation for MSG_RCV and RCV in α. ∎

The following lemma states several properties of executions of p_i.

Lemma 8.13: Let α be an execution of p_i containing states s' and s, in that order. Then $\forall j \in \mathcal{I}$,

1. If $s'.$*state-snapped* = true, then $s.$*state-snapped* = true.
2. If $s'.$*chan-snapped*(j) = true, then $s.$*chan-snapped*(j) = true.
3. If $s'.$*done* = true, then $s.$*done* = true.
4. If $s.$*chan-snapped*(j) = true, then $s.$*state-snapped* = true.
5. If $s'.$*state-snapped* = true, then $s.$*snapshot* = $s'.$*snapshot*.
6. If $s'.$*chan-snapped*(j) = true, then $s.$*chan-state*(j) = $s'.$*chan-state*(j).
7. If START$_i$ occurs before state s, then $s.$*state-snapped* = true.
8. If MSG_RCV$(\#,j,i)$ occurs before state s, then $s.$*chan-snapped*(j) = true.
9. If *state-snapped* = false, then *chan-state*(j) = ϵ.

Proof: Properties 1-3 are immediate from inspection of the code for p_i, since no action of p_i sets those variables to false. Property 4 follows from Property 1 and the definition of MSG_RCV, the only action that can set a *chan-snapped* variable to true. Property 5 follows from the definitions of START$_i$ and MSG_RCV, which only modify *snapshot* if *state-snapped* is false. Similarly, Property 6 follows from the fact that a MSG_RCV action only modifies *chan-state*(j) if *chan-snapped*(j) is false. Property 7 follows from the definition of START$_i$ and Property 1. Property 8 follows from the definition of MSG_RCV and Property 2. Since *chan-state*(j) is initially empty and only modified by a MSG_RCV action when *state-snapped* = true, Property 9 follows from Property 1. ■

In the following lemma, we use the above stable properties and invariants to show exactly what p_i reports in a DONE$_i$ action.

Lemma 8.14: Let α be an execution of p_i containing a DONE$_i(a, c)$ action, and let s be the first state of α in which *state-snapped* = true. Then $a = s.app$, and for all $j \in \mathcal{I}$, $c(j)$ contains the sequence of messages m appearing in all the MSG_RCV(m, j, i) actions between state s and the first state s' in which *chan-snapped*(j) = true.

Proof: From the precondition on DONE$_i$ we know that state s must exist. The only actions that can cause *state-snapped* to become true are the START$_i$ and MSG_RCV actions. When either of these actions sets *state-snapped* = true, they also copy *app* into *snapshot*. Therefore, $s.snapshot = s.app$. So, from Properties 1 and 5 of Lemma 8.13, we know that this value of *snapshot* remains fixed for the remainder of α. Therefore, by the definition of DONE$_i$, $a = s.snapshot = s.app$.

For all $j \in \mathcal{I}$, we know from Property 9 of Lemma 8.13 that in all states s' before s in α, $s'.chan\text{-}state[j] = \epsilon$. When the first RCV_MSG$(\#, j, i)$ occurs in α, *chan-snapped*(j) is set to true, and *chan-state*(j) becomes fixed by Properties 2 and 6. Then by the definition of MSG_RCV, the sequence of messages m appearing in all the MSG_RCV(m, j, i) actions between state s and the first state s' in which *chan-snapped*(j) = true is exactly the sequence of messages added to *chan-state*(j) in α, and they are added in the order of occurrence in α. Therefore, by definition of DONE$_i$, the lemma holds. ■

8.2.3 Proof of Correctness

Throughout the proof, we use subscripts to distinguish the state components of different processes. For example, app_i refers to the *app* component of p_i.

Let automaton P be the composition of all p_i, $i \in \mathcal{I}$, and let X be the set of variables app_i, $i \in \mathcal{I}$. Let $Q = \mathcal{U}(P, X)$. We wish to show that Q solves partial execution module G. First, we prove a statement about interprocess communication in Q, and then turn directly to the main result.

Lemma 8.15: Let α be an admissible fair extended execution of Q. Then there exists a live correspondence relation for SEND and RCV in α.

Proof: Since α is admissible, we know there is a live correspondence relation between MSG_SEND and MSG_RCV in α. Therefore, by Lemmas 8.12 and 8.10, we have the desired result. ∎

Theorem 8.16: Automaton Q solves partial execution module G.

Proof: The organization of the proof follows the definition of G. Let γ be an admissible fair extended execution of Q, and let $\alpha = \gamma|(acts(G), X)$. For condition (1) of G, we wish to show that $sched(\alpha)|U \in fairbehs(S_{app})$. From the hypothesis ($\alpha$ admissible), we know that for all $i \in \mathcal{I}$, $\alpha|u_i \in fairbehs(u_i)$. Therefore, by Lemma 2.1, we know that $\alpha \in fairexecs(U)$. And from Lemma 8.15, we know that $sched(\alpha)$ is network admissible. Therefore, $sched(\alpha)|U \in fairbehs(S_{app})$.

For condition (2) of G, suppose that a START action occurs in α. For (2a), we wish to show that exactly one DONE_i occurs in α for each $i \in \mathcal{I}$. For each $i \in \mathcal{I}$ when the first START_i or MSG_RCV(#,j,i), $j \in \mathcal{I}$, action occurs, a marker (#) is placed into every *out-queue*[j] $\in$ *out-chans*(i). Therefore, if a START_i occurs or p_i receives a marker, then for all $j \in$ *out-chans*(i), a MSG_SEND(#,i,j) eventually occurs. Since α is admissible, a MSG_RCV(#,i,j) eventually occurs for all $j \neq i$. We know that the graph CHANS is strongly connected. Thus, if a START_i occurs in α, then eventually a marker is received by each p_i on each of its incoming channels. The first time a START_i occurs or a marker is received at p_i, the state of u_i is recorded. Furthermore, when a MSG_RCV(#,j,i) occurs, p_i records the state of incoming channel j. Therefore, it is eventually the case that for all $i, j \in \mathcal{I}$, $state\text{-}snapped_i$ = true and

chan-snapped$[j]$ = true. By Properties 1 and 2 of Lemma 8.13, we know that these are stable properties. Therefore, since the preconditions on DONE$_i$ eventually become true and remain true until it occurs, a DONE$_i$ action must eventually occur in α for each $i \in \mathcal{I}$. By Property 3 of Lemma 8.13 at most one DONE$_i$ action can occur in α for each $i \in \mathcal{I}$, since that action sets *done*$_i$ to true. This completes the proof of part (2a).

We prove part (2b) by construction. For all $i \in \mathcal{I}$, let s_i^* be the first state in α in which *state-snapped*$_i$ = true. (We have already shown that such a state must exist.) Now, for all $i \in \mathcal{I}$, mark all actions of $\alpha|u_i$ after s_i^* as *distinguished.* Note that all actions in α_1 are not distinguished, all actions in α_3 are distinguished, and α_2 contains a mixture of distinguished and undistinguished actions. We construct $\beta = \beta_1\beta_2 s \beta_3\beta_4$ as follows:

1. Let $\beta_1 = \alpha_1|U$.
2. Let β_2 (β_3) contain the sequence of undistinguished (distinguished) actions of U in α_2, where each action π is followed by state $s_\beta \in \mathit{states}(U)$ such that for all $i \in \mathcal{I}$,
   (a) if $\pi \in \mathit{acts}(u_i)$ then $s_\beta|u_i = s_\alpha|u_i$, where s_α is the state following π in α, and
   (b) if $\pi \notin \mathit{acts}(u_i)$ then $s_\beta|u_i = s'_\beta|u_i$, where s'_β is the previous state in β.
3. Let $\beta_4 = \alpha_3|U$.

Informally, we construct β from $\alpha|U$ by "delaying" the actions of a process that has recorded its local snapshot until all the remaining processes have also recorded theirs. The sequence β_1 is the prefix of $\alpha|U$ up to the first START action. The sequence β_2 contains all the remaining actions of $\alpha|U$ for processes that have not yet taken their local snapshots. The sequence β_3 contains all the "delayed" actions, up until the last process reports its snapshot. Finally, β_4 is the suffix of $\alpha|U$ after the global shapshot has been completed.

Clearly, for all $i \in \mathcal{I}$, $\beta|u_i = \alpha|u_i$. Therefore, by Lemma 2.1, β is an execution of U. Next, we need to show that there exists a live correspondence relation C_β for SEND and RCV in β. We will show, in fact, that it is the *same* correspondence relation as in α. We know that the same actions occur in β as in α. Therefore, the only condition we need to show is that for each SEND action, the corresponding RCV occurs later in β:

Suppose (for contradiction) that there exists RCV(m, i, j) in β such that the corresponding SEND(m, i, j) occurs later in β. The only way this could happen is for the RCV to be an

undistinguished action in α and the SEND to be a distinguished action in α, or else they could not have been reordered by the construction. However, if the SEND is a distinguished action, then the message from that SEND must be preceded in the outgoing channel by a marker message from u_i to u_j, so the RCV for the marker occurs at the u_j before the RCV for m, after which *state-snapped*$_j$ = true. This means that any later actions of u_j are distinguished, a contradiction. Therefore, the live correspondence relation $\mathcal{C}_\beta$ exists.

We now consider each of the four properties in condition (2b). Property (i) holds immediately from the construction, since the construction preserves the order of events at each automaton u_i, $i \in \mathcal{I}$. We know that α_1 is the prefix of α up to the first START action. Since no process p_i sets *state-snapped*$_i$ = true until after the first START action occurs, we know from the construction that $\alpha_1|U = \beta_1|U$. Therefore, the Property (ii) holds. Since $\beta_1\beta_2$ contains exactly the sequence of undistinguished actions in α, we know that for all $i \in \mathcal{I}, s|u_i$ is the state of u_i in α when *state-snapped*$_i$ first becomes true. Moreover, we know that for all $i \in \mathcal{I}$, in-transit$_{\beta_1\beta_2,\mathcal{C}_\beta}(i)$ maps each $j \in \mathcal{J}$ to the set of all messages sent by u_j before *state-snapped*$_j$ = true, but not received by u_i before *state-snapped*$_i$ = true. Whenever *state-snapped*$_j$ becomes true for the first time, p_j places a marker in all outgoing channels. Therefore, for all $i \in \mathcal{I}$, in-transit$_{\beta_1\beta_2,\mathcal{C}_\beta}(i)$ maps each $j \in \mathcal{J}$ to exactly the sequence of messages m appearing in all the MSG_RCV(m, j, i) of α between the first state in which *state-snapped*$_i$ = true and the last state before a marker message is received by p_i along the channel from p_j. Since receipt of a marker message from p_j results in *chan-snapped*$_i(j)$ = true (Property 8 of Lemma 8.13), we know from Lemma 8.14 that Property (iii) holds. From the construction $\beta_4 = \alpha_3|U$. Since the set of actions in $\alpha|U$ is exactly the set of actions in β and the correspondence relations for SEND and RCV are the same, Property (iv) holds. ∎

This completes the correctness proof for the global snapshot algorithm.

8.3 Supporting Superposition in Spectrum

To conclude the chapter, we propose extensions to the Spectrum Simulation System for supporting superposition.

Our general approach is to permit automata to refer to the states of other automata by

```
DATA message string
DATA queuemap mapping(automaton_id, sequence(message))

AUTOMATON app
   STATE ...

AUTOMATON snapshot
   STATE tuple(state-snapped:boolean, chan-snapped:boolean,
               in-queue:queuemap, out-queue:queuemap, chan-state:queuemap,
               temp:queuemap
               snapshot: type(app),
               child: automaton_id(app))
   INPUT Start
      EFF ifthen(eq(s.state-snapped,false), {
                    assign(s.snapshot, state(s.child))
                    assign(s.state-snapped,true)
                    forall_do(k, out(self()), {
                      assign(s.temp, map_eval(out_queue,k))
                      seq_addb(s.temp, "SS")
                      map(out-queue, k, s.temp)
                      })
                    })
```

Figure 8-2: Spectrum automaton type definition for an unconstrained automaton.

automaton id using two special operators. The operator `type(x)`, where `x` is the name of an automaton type, refers to the *data type* of states of automata of type `x`. The operator `state(i)`, where `i` is an automaton id, refers to the state of the automaton with id `i`. One is permitted to refer to such state information for any lower level automata in the superposition hierarchy, which we explain shortly. In keeping with the definition of an unconstrained automaton, we have the loader enforce the requirement that one may not modify the state of another automaton.

In order to provide type checking on references to states of other automata, we convert our one-pass loader into a two-pass loader. In the first pass, all DATA declarations and automaton type STATE declarations are processed. Then, in the second pass, the rest of the file is processed and information gained in the first pass may be used in processing an automaton type definition to fill in the type information for references to the states of other automata. As a requirement for this type checking, we must know, when the state of an automaton is referenced, the type of that automaton. Therefore, we modify the language so that all automaton id's are typed, as opposed to the current definition in which all automaton id's are considered to have the same type.

The code fragment in Figure 8-2 illustrates Spectrum syntax for superposition, and corresponds to a piece of the global snapshot protocol in Figure 8.2.2.[5] We assume that an automaton type `app` has been defined as the application automaton type. In addition, we assume that the system is configured so that each automaton of type `snapshot` is superposed on exactly one automaton of type `app`, and that the `child` component of the state of each `snapshot` automaton is set to the automaton id of the corresponding `app` automaton in the initially action. In the incomplete definition for the `snapshot` automaton, notice that the state component `snapshot` has the same type as the state of automata of type `app`. That is, the `type()` operator lets us say that no matter how the state of `app` is declared, this component of the `snapshot` automaton has the identical data type. Therefore, it is not necessary to modify the `snapshot` automaton type definition whenever the state declaration of the underlying application is changed. Notice that automaton id's are now typed: for example, the state component `child` is defined to be of type `automaton_id(app)`, and not just of type `automaton_id`. The use of the `state()` oper-

[5] We use "SS" for the snapshot marker instead of "#" because only alphanumerics are permitted in Spectrum strings.

ator in the second line of the EFF clause is straightforward. It simply refers to the state of the underlying application automaton. In the loader, the type of this expression is simply deduced from the automaton type of the argument. If necessary, we may refer to pieces of the state of the underlying module using the usual record notation, if the data type of the state is a tuple. For example, one might write `state(s.child).foo`.

In a given step, we want the higher level modules to use the "prior" states of the underlying modules in taking their steps. Therefore, the order of evaluation is important: the execution loop must ensure that within a given step, the new states of the automata are evaluated top down in the superposition hierarchy, and that the new set of enabled classes is evaluated only after all state changes have been made.

The superposition operator, like composition and closeout, is handled by the user interface. In the loader, each automaton type that refers to the state of an underlying component, is flagged as an unconstrained automaton. When such an automaton is opened in the types menu, the user interface presents an auxiliary window that contains the data types of the state components at the top. Below this information is a work area, like that in the main window or in auxiliary windows for composed types, in which one may specify the set of underlying components in the superposition, just as one specifies the set of components of a composition. So that an automaton may discover its set of underlying components, we add to the language a configuration data function `children(x,t)`, where `x` is an automaton id and `t` is an automaton type name, that returns the set of automaton id's of those underlying components of the superposed automaton `x` that have automaton type `t`. For example, `children(self(),app)` would have been used in the snapshot automaton described above to discover the id of the underlying application automaton.

8.4 Summary

In this chapter, we extended the I/O automaton model to permit superposition of program modules. As an illustration of the extended model, the global snapshot algorithm of Chandy and Lamport was presented with a complete proof of correctness. Finally, we proposed extensions to the Spectrum Simulation System to support the extended model.

Chapter 9

Distributed Simulation

In Chapter 6, we mentioned that one way to improve a simulation system designed for experimenting with algorithms is simply to make the simulator run faster. Optimizations in the sequential simulator can help in this regard, but one way to dramatically increase the simulation speed is to introduce concurrency in the simulator. Since this thesis is concerned with distributed algorithms, in this chapter we present an algorithm that could be used to achieve highly concurrent distributed simulation of I/O automaton systems. We present a new synchronization problem that arises from the particular semantics of the I/O automaton model, and we present a highly concurrent protocol to solve the problem. The problem statement, protocol, and correctness proof are all formally stated using the I/O automaton model.

We consider a set of n processes in an asynchronous system whose computation proceeds by a sequence of *multicasts* (or *partial broadcasts*). In each multicast, a process u sends a message m to an arbitrary subset S of the processes (including u). We say that a protocol solves the *logically synchronous multicast* problem if it guarantees the following conditions:

(1) There exists a total order on all multicasts in a computation such that the delivery order of multicast messages at each process is consistent with that total order.

(2) If process u sends message m, it receives no messages between sending and receiving m.

(3) If process u continually wishes to send a message, then eventually u will send a message.

The first two conditions say that it appears to all processes as if each multicast occurs simultaneously at all of its participants (sender plus receivers). Hence, the name *logically synchronous*

multicast. Note that the hypothesis of the third condition does not require that u continually wish to send the *same* message, but only *some* message. This is a technical point that will be of importance later.

The problem lends itself to a highly concurrent solution, since any number of multicasts with disjoint S sets should be able to proceed independently. Likewise, one would expect that the communication costs of an algorithm to solve this problem would be independent of n. We present a solution that takes advantage of the concurrency inherent in the problem and requires at most $4|S|$ messages per multicast, provided that a process does not "change its mind" about the set of participants.

Various approaches to ordering messages in asynchronous systems have been studied. Lamport [38] uses logical clocks to produce a total ordering on messages. Birman and Joseph [7] present several types of fault-tolerant protocols, where failures are assumed to be detectable by timeouts. Their ABCAST (atomic broadcast) protocol guarantees that broadcast messages are delivered at all destinations in the same relative order, or not at all. Their CBCAST (causal broadcast) protocol provides a similar, but slightly weaker, ordering guarantee to achieve better performance. The CBCAST guarantees that if a process broadcasts a message m based on some other message m' it had received earlier, then m will be delivered after m' at all destinations they share. Broadcast protocols may be used to achieve process synchronization in distributed systems. For example, Schneider presents a synchronization technique that assumes a process may reliably broadcast a message to all other running processes such that messages originating at a given process are received by other processes in the order sent [57]. Joseph and Birman provide an extensive discussion of reliable broadcast protocols in [33].

Like ours, the protocols of both [38] and [7] assign a global ordering to messages. However, these protocols do not solve the logically synchronous multicast problem because they allow messages to "cross" each other. That is, in their protocols a process u may send a message m and at some time later receive a message ordered before m. Our problem requires that when a process u sends a message m, it must have "up to date" information, meaning that it has already received all messages destined for u that are ordered before m. (See Condition (2) above.)

Motivated by CSP [31] and ADA [1], multiway handshaking protocols have been studied

extensively. (For examples, see [5], [6], and [12].) These protocols must enforce a very strict ordering on system events, and therefore achieve less concurrency (than ours and the others mentioned above). This is necessary because the models of CSP and ADA permit processes to block inputs. Since a decision about whether to accept or refuse an input may depend (in general) on all earlier events, each process can allow the scheduling of only one event (input or output) to proceed at a time. That is, process p_1 cannot permit process p_2 to complete an event e until p_1 knows that no event e' to be ordered before e will cause e to be refused by p_1. In general, p_1 cannot permit p_2 to complete e until all events at p_1 ordered before e have already occurred. Our problem admits more concurrency, since processes may schedule multiple input events at a time. That is, a process p can permit many multicasts destined for p to proceed concurrently, since p cannot refuse any of them, regardless of the orderings.[1]

One interesting feature of our problem is that it lies between the two general approaches described above. As we have described, it permits concurrent scheduling of events, yet imposes a strong, useful structure on the message delivery order.

Other related work includes papers by Awerbuch [4] and Misra [49], which study different problems in the area of simulating synchronous systems on asynchronous ones. In both cases, however, the computational models being simulated are very different from ours. Awerbuch's goal is to take algorithms written for systems in which processes proceed in lock step, and simulate them on systems in which processes proceed asynchronously. An algorithm is presented for generating "pulse" messages to synchronize the computation. In contrast, the purpose of logically synchronous multicast is to provide the illusion of synchronous communication among dynamically changing subsets of processes, as opposed to synchronized steps at all processors. Misra [49] studies the problem of distributed discrete event simulation. The essential difference between Misra's work and logically synchronous multicast is that Misra fixes the communication pattern. This gives the problem additional structure, since each process expects messages only from a (small) fixed subset of the other processes. In the present work, we assume that a process may potentially receive a multicast from any other process in the system. In spite of this difference, some of Misra's techniques, particularly those for breaking deadlock, can be

[1] These comments apply only to *pessimistic* protocols, in which no rollback is allowed. If rollback is permitted, an *optimistic* strategy for CSP-style synchronization could be achieved with more concurrency, but at the expense of the overhead necessary for rollback.

applied to our problem. This is discussed in Section 9.4.3.

The remainder of the chapter is organized as follows. In Section 9.1, we present the architecture of the logically synchronous multicast problem and a statement of correctness in terms of the model. In Section 9.2, we formally present the algorithm using the I/O automaton model. In Sections 9.3 and 9.4, we give a complete correctness proof and analyze the message and time complexities. We conclude the chapter by describing how the logically synchronous multicast protocol could be used to achieve distributed simulation of I/O automaton systems.

9.1 The Problem

In this section, we describe the architecture of the logically synchronous multicast problem and then present a schedule module to define correctness for a multicast protocol.

9.1.1 The Architecture

Let $\mathcal{I} = \{1, \ldots, n\}$. Let $\mathcal{S}$ denote a universal set of text strings (containing the empty string ϵ), and let $\mathcal{M}$ denote a universal set of messages. Let $u_i, i \in \mathcal{I}$, denote the n user processes engaged in the computation, and let $p_i, i \in \mathcal{I}$, denote n additional processes. Together, the p_i's are to solve the multicast problem, where each p_i is said to "work for" u_i. Each of the u_i's and p_i's is modelled as an automaton.

Each user u_i directly communicates by shared actions with the process p_i only. (One may think of u_i and p_i as running on the same processor.) The p_i's communicate with each other asynchronously via a network, also modelled as an automaton, that guarantees eventual one-time delivery of each message sent. Furthermore, we assume that all messages sent between each pair of processes are delivered in FIFO order.

The boundary between u_i and p_i is defined by several actions. To summarize the relationship between u_i and p_i at each point in an execution, we say that p_i is in a certain *region*, according to which of these actions has occurred most recently. We will formalize this later. Figure 9-1 illustrates the actions shared by u_i and p_i, and by p_i and the network. Figure 9-2 illustrates the possible region changes for p_i, and the actions that cause them.

Initially, p_i is in its "passive" region (P). We say that p_i enters its "trying" region (T)

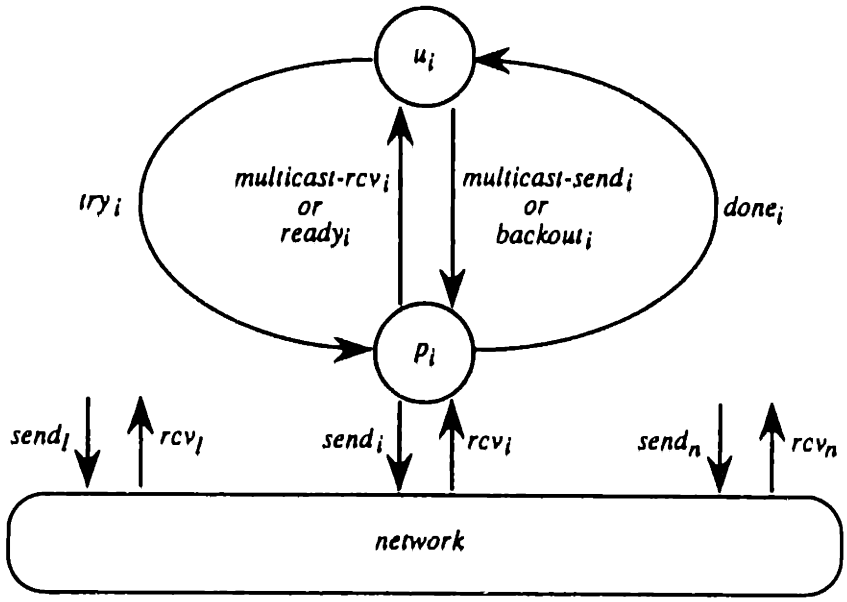

Figure 9-1: System Architecture. Arguments of actions are omitted.

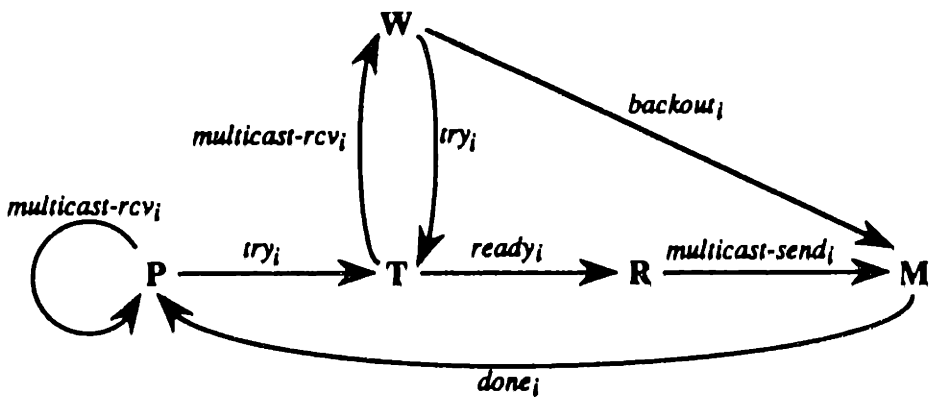

Figure 9-2: Region Changes for p_i.

when user u_i issues a $try_i(S \subseteq \mathcal{I})$[2] action, indicating that u_i would like to send a multicast message to processes named in the set S. When it is ready to perform a multicast on behalf of u_i, process p_i issues a $ready_i$ action and is said to enter its "ready" region (R). The $ready_i$ action constitutes permission for u_i to actually send the multicast. That is, after receiving the $ready_i$ action as input, user u_i may issue a $multicast\text{-}send_i(m \in \mathcal{S})$ action, where the argument indicates the desired text of the multicast message. Upon receiving the $multicast\text{-}send_i$ action, p_i is said to enter its "multicast" region (M), where it completes the multicast and returns to region P by issuing a $done_i$ action. Region M is present to ensure that each multicast for u_i is completed before the next multicast is requested by u_i.

In addition to these actions, there are $multicast\text{-}rcv_i(m \in \mathcal{S})$ actions, which are outputs of p_i and inputs to u_i. The purpose of these actions, which may occur while p_i is in P or T, is to forward multicast messages to u_i that were sent to p_i by some process p_j on behalf of user u_j. The argument m is the text of the multicast message. To correspond with this additional type of action, we have a "waiting" region (W), which is entered whenever p_i issues a $multicast\text{-}rcv_i$ action while in T.[3] In W, p_i waits to see if u_i has "changed its mind" about its own multicast after hearing the information contained in the $multicast\text{-}rcv_i$ action. Either u_i still wishes to perform some multicast and issues a $try_i(S')$ action, or u_i decides not to do a multicast after all and issues a $backout_i$ action.

It might seem that one could eliminate region W and the $backout_i$ actions by having $multicast\text{-}rcv_i$ actions take p_i to region P. However, this would make it difficult to express the liveness notion that u_i eventually must be allowed to perform a multicast, provided that it continually wants to do so. Region W is used to signify that u_i has a choice of continuing to try or "giving up." As a separate modification of this architecture, one might consider elimination of the $ready_i$ and $multicast\text{-}send_i$ actions in favor of including the desired text of the multicast as a a second argument to the try_i actions. However, as we will see, the $ready_i$ and $multicast\text{-}send_i$ actions serve as useful "commit" points in stating both the safety and liveness conditions of the problem. They also provide a convenient way to separate the successful multicasts from the unsuccessful try_i attempts in reasoning about algorithm executions.

[2] That is, $try_i(S)$, where $S \subseteq \mathcal{I}$.

[3] A $multicast\text{-}rcv_i$ action from region P does not cause a region change.

9.1.2 Correctness

Since the only actions under the control of the protocol are the outputs of the p_i's, we only wish to require that the protocol behaves correctly when its environment, namely the composition of the u_i's and the network, is well-behaved. To this end, we define schedule modules that specify the allowable behaviors of each u_i and the network. Based on these, we define a schedule module for the multicast protocol. We begin with the schedule modules for the u_i's.

Schedule Module U_i: We define the signature of U_i as follows:

$\text{in}(U_i) = \{multicast\text{-}rcv_i(m \in \mathcal{S}),\ ready_i,\ done_i\}$

$\text{out}(U_i) = \{try_i(S \subseteq \mathcal{I}),\ multicast\text{-}send_i(m \in \mathcal{S}),\ backout_i\}$

Before defining the set of schedules of U_i, we define a "region sequence" to capture the series of region changes in a schedule, and then state a *well-formedness* condition which makes use of this definition. Let the alphabet $\Sigma = \{\text{P, T, R, M, W, X}\}$. Let α be an arbitrary sequence of actions. We define the *region of i after α*, denoted $r(i, \alpha)$, to be an element of Σ defined recursively as follows. If $\alpha|U_i$ is empty (ϵ), then $r(i, \alpha) = \text{P}$. If $\alpha = \alpha'\pi$, then, ignoring arguments to action names,

$$r(i,\alpha) = \begin{cases} r(i,\alpha') & \text{if } \pi \notin acts(U_i), \\ \text{P} & \text{if } (\pi = done_i \wedge r(i,\alpha') = \text{M}) \vee (\pi = multicast\text{-}rcv_i \wedge r(i,\alpha') = \text{P}), \\ \text{T} & \text{if } \pi = try_i \wedge r(i,\alpha') \in \{\text{P,W}\}, \\ \text{R} & \text{if } \pi = ready_i \wedge r(i,\alpha') = \text{T}, \\ \text{M} & \text{if } (\pi = multicast\text{-}send_i \wedge r(i,\alpha') = \text{R}) \vee (\pi = backout_i \wedge r(i,\alpha') = \text{W}), \\ \text{W} & \text{if } \pi = multicast\text{-}rcv_i \wedge r(i,\alpha') = \text{T}, \\ \text{X} & \text{otherwise.} \end{cases}$$

Given an arbitrary action sequence α and an index $i \in \mathcal{I}$, we define the *region sequence for i in α*, denoted *region-sequence*(i, α), to be the concatenation of $r(i, \alpha')$ for each prefix of α in order, starting with $r(i, \epsilon)$ and ending with $r(i, \alpha)$. Note the close correspondence between Figure 9-2 and the definition of region-sequence.

Let α be an arbitrary sequence of actions. We say that α is *user well-formed for i* iff

1. for all $try_i(S)$ actions in α, $i \in S$, and

2. *region-sequence*(i, α) does not contain the symbol X.

We can now define the set of schedules for U_i. Let α be a sequence of actions in sig(U_i). Then $\alpha \in$ *scheds*(U_i) iff

1. U_i preserves user well-formedness for i in α, and

2. region-sequence(i, α) does not end in W or R.

The first property is used to help define the safety conditions for the logically synchronous multicast problem, since a multicast protocol must perform correctly only if its environment is well behaved. The second property, used in defining the liveness conditions, says that a user process cannot "stop" in regions W or R. This is used to express the notion that a multicast protocol must guarantee progress only if users trying to send multicasts eventually respond to *multicast-rcv* and *ready* actions.

We define schedule module U to be the composition $\Pi_{i \in \mathcal{I}} U_i$.

Schedule Module N: We now define a schedule module specifying the network. The signature is as follows:

$\text{in}(N) = \{send(m \in \mathcal{M}, i, j \in \mathcal{I})\}$

$\text{out}(N) = \{rcv(m \in \mathcal{M}, i, j \in \mathcal{I})\}$

To define the allowable schedules of the network, we use a *correspondence relation* similar to that of Fekete and Lynch [17]. A correspondence relation between the *send* and *rcv* events in a sequence captures the correspondence between the send and receipt of a message. Consider the following properties that may hold for a particular correspondence relation for a given sequence α:

(S1) $\forall i_1, i_2, j_1, j_2 \in \mathcal{I}, \forall m_1, m_2 \in \mathcal{M}$, if event $\pi_1 = send(m_1, i_1, j_1)$ corresponds to event $\pi_2 = rcv(m_2, i_2, j_2)$, then $m_1 = m_2$, $i_1 = i_2$, $j_1 = j_2$, and π_1 precedes π_2 in α.

(S2) $\forall i, j \in \mathcal{I}, \forall m \in \mathcal{M}$, each $rcv(m, i, j)$ corresponds to exactly one $send(m, i, j)$.

(S3) $\forall i, j \in \mathcal{I}, \forall m \in \mathcal{M}$, each $send(m, i, j)$ corresponds to at most one $rcv(m, i, j)$.

(S4) $\forall i, j \in \mathcal{I}, \forall m, m' \in \mathcal{M}$, if event $rcv(m, i, j)$ occurs in α before event $rcv(m', i, j)$, then their corresponding events $send(m, i, j)$ and $send(m', i, j)$ occur in the same order.

(L) $\forall i, j \in \mathcal{I}, \forall m \in \mathcal{M}$, each $send(m, i, j)$ event has a corresponding $rcv(m, i, j)$ event.

The first four properties (S1–S4) are *safety* properties. They say that a message is delivered only after it is sent, that no spurious messages are delivered, that a message is delivered at most once (for each time it is sent), and that messages between a pair of processes are delivered in the order sent. Property (L) is a *liveness* property; it says that each message sent is eventually delivered.

If α is a sequence of actions of N, we say that α is *network well-formed* iff there exists a correspondence relation for α that satisfies properties S1–S4. Moreover, $\alpha \in scheds(N)$ iff the correspondence relation also satisfies property (L). Property (L) will be used only in the liveness proof.

Schedule Module M: The correctness conditions for the logically synchronous multicast problem can now be stated formally in terms of the actions at the boundaries of the user processes. We do this with a schedule module M that defines the multicast problem. We define the signature of M as follows:

$\text{in}(M) = \text{out}(U) \cup \text{out}(N)$

$\text{out}(M) = \text{in}(U) \cup \text{in}(N)$

In defining the schedules of M, we use a correspondence relation technique (similar to the one used to define schedule module N) to capture the correspondence between each *multicast-send* action and the resulting *multicast-rcv* actions. Let α be a sequence of actions of $\text{sig}(M)$, and let correspondence relation C relate the *multicast-send* and *multicast-rcv* actions of α. We say that C is a *proper correspondence relation for* α iff it satisfies the following properties:

1. $\forall i, j \in \mathcal{I}, \forall m, m' \in S$, if event $\pi_1 = multicast\text{-}send_i(m)$ corresponds to event $\pi_2 = multicast\text{-}rcv_j(m')$, and $try_i(S)$ is the last try_i action in α before π_1, then $m = m'$ and $j \in S$.

2. $\forall i, j \in \mathcal{I}, \forall m \in S$, each $multicast\text{-}rcv_j(m)$ corresponds to exactly one $multicast\text{-}send_i(m)$.

3. $\forall i, j \in \mathcal{I}, \forall m \in \mathcal{S}$, each *multicast-send*$_i(m)$ corresponds to at most one *multicast-rcv*$_j(m)$.

Informally, these properties say that (1) a *multicast-rcv*$_j(m)$ must contain the same text argument as its corresponding send, and that j must name one of the destination processes, (2) a *multicast-rcv* action corresponds to exactly one *multicast-send*, and (3) a given *multicast-send* action corresponds to at most one *multicast-rcv*$_j$ for each possible destination process u_j.

Let α be a sequence of actions of sig(M), let $\mathcal{C}$ be a proper correspondence relation for α, and let $\prec$ be a total order on all *multicast-send* actions in α. We say that $\prec$ is a *proper total order for* $\mathcal{C}$ *and* α iff the following property holds: $\forall i, j, k \in \mathcal{I}, m, m' \in \mathcal{S}$, if *multicast-send*$_i(m)$ and *multicast-send*$_j(m')$ occur in α with corresponding receives *multicast-rcv*$_k(m)$ and *multicast-rcv*$_k(m')$, and if $\prec$ orders *multicast-send*$_i(m)$ before *multicast-send*$_j(m')$, then *multicast-rcv*$_k(m)$ occurs in α before *multicast-rcv*$_k(m')$. Informally, this says the order of multicast deliveries at each user process must be consistent with the total order $\prec$.

Let α be a sequence of actions of sig(M). Then $\alpha \in$ *scheds*(M) iff there exists a correspondence relation $\mathcal{C}$ and total order $\prec$ such that the following conditions hold.

1. $\forall i \in \mathcal{I}$, M preserves user well-formedness for i in α.

2. If α is user well-formed for every $i \in \mathcal{I}$ and α is network well-formed, then

   (a) $\mathcal{C}$ is a proper correspondence relation for α,

   (b) $\prec$ is a proper total order for $\mathcal{C}$ and α, and

   (c) $\forall m \in \mathcal{S}$, if $\pi =$ *multicast-send*$_i(m)$ occurs in α, then no *multicast-rcv*$_i$ occurs between π and the *multicast-rcv*$_i(m)$ corresponding to π.

3. If $\alpha|N \in$ *scheds*(N) and $\forall i \in \mathcal{I}$, $\alpha|U_i \in$ *scheds*(U_i), then the following hold:

   (a) $\forall i \in \mathcal{I}$, if a *try*$_i$ occurs in α, then either a *backout*$_i$ or a *ready*$_i$ occurs later in α.

   (b) $\forall i \in \mathcal{I}, S \subseteq \mathcal{I}$, if a *multicast-send*$_i(m)$ occurs in α and *try*$_i(S)$ is the last preceding *try*$_i$ action in α, then a corresponding *multicast-rcv*$_j(m)$ occurs later in α for each $j \in S$.

Items (1) and (2) are the required *safety* properties. Part (2c) is needed to ensure that user processes have "up to date" information when sending a multicast message. Item (3) is

the required *liveness* property. Part (3a) says that if a user process does not back out of its attempt to perform a multicast, then eventually it will receive permission to send the multicast. Part (3b) says that if a multicast is sent by a user process, then eventually all destination user processes will receive it. Note that the hypothesis of item (3) is needed to ensure that liveness properties hold for the users and the network. That is, we require that a solution to the multicast problem guarantee progress only if the users and the network satisfy their liveness requirements, namely that every user responds to *multicast-rcv* and *ready* actions and that every message is eventually delivered. A multicast protocol is *correct* iff it solves M.

9.2 The Algorithm

This section presents the multicast protocol. We present the algorithm by giving an explicit I/O automaton for each p_i, $i \in \mathcal{I}$. We show in Section 9.3 that the composition of the p_i's solves the schedule module M and is therefore a correct protocol.

The algorithm is based on logical time. We define a *logical time* to be an (integer, process-id) pair drawn from $\mathcal{T} = (\{1, 2, \ldots\} \cup \infty) \times \mathcal{I}$, and we let logical times be ordered lexicographically. Essentially, each process p_i maintains a logical time clock, and each multicast is assigned a unique logical time.[4] The process p_i delivers all multicast messages destined for u_i in logical time order.

The state of each automaton p_i has several components. The variable *region* $\in$ {P,T,W,R,B} is initially set to P and holds the current region of p_i, as described in Section 9.1.1. The variables *try-set*, *need-set*, *requested*, and *requests* are subsets of $\mathcal{I}$, initially empty. The *try-set* names the processes to whom u_i would like to send a multicast, and the *need-set* contains the union of all values of *try-set* since the last multicast by p_i (or the beginning of the execution). The two sets *requested* and *requests* name the processes to whom p_i has sent requests for "promises" and the processes from whom p_i has received such requests. We will explain promises shortly. The variable *text* $\in S$ is initially undefined, and is used to hold the text of the latest multicast by u_i. Two arrays of logical times indexed by $\mathcal{I}$ are kept: *promises-to* and *promises-from*. The entries of these arrays, initially (∞, n), are used to keep track of the times of promises granted

[4] We never use ∞ in the time of a multicast message; it is used only as a place holder.

and received, respectively. Two additional logical time variables, *clock* and *mctime*, are initially $(0, i)$. The *clock* contains the current logical time of u_i's computation, and *mctime* contains the time of the latest multicast by u_i. Finally, the variable *pending* is an initially empty set of (text $\in \mathcal{S}$, time $\in \mathcal{T}$) pairs. This set contains all multicast messages received by p_i but not yet delivered to u_i.

We let min(*promises-to*) denote the smallest time among the entries in the *promises-to* array. Similarly, we let max(*promises-from*) denote the largest time less than (∞, n) among the entries in the *promises-from* array; if all entries in that array are (∞, n), then max(*promises-from*) = $(0, i)$. Finally, we let min(*pending*) and max(*pending*) denote the pairs in the *pending* set having the least and greatest logical times, respectively; if *pending* is empty, then both values are $(\epsilon, (0, i))$.

The transition relation for p_i is shown in Figure 9-3. "P" and "E" denote precondition and effect, respectively. An action is enabled in exactly those states s' for which the precondition is satisfied. If an action has no precondition, it is enabled in all states. When an action occurs, p_i's new state s is determined according to the statements in the effects clause. States s and s' agree on components not assigned values in the effects clause. Automaton p_i has the following signature.

Input actions:	$try_i(S \subseteq \mathcal{I})$	Output actions:	$multicast\text{-}rcv_i(m \in \mathcal{S})$
	$backout_i$		$ready_i$
	$multicast\text{-}send_i(m \in \mathcal{S})$		$done_i$
	$rcv(m \in \mathcal{M}, j \in \mathcal{I}, i)$		$send(m \in \mathcal{M}, i, j \in \mathcal{I})$

The equivalence classes of $part(p_i)$ are as follows. The actions $multicast\text{-}rcv_i$, $ready_i$, and $done_i$ are together in one class. And for each $j \in \mathcal{I}$, there exist four classes containing the sets of actions $send$(promise($t \in \mathcal{T}$),i, j), $send$(req-promise,i, j), $send$(adv-promise($t \in \mathcal{T}$),i, j), and $send$(multicast($m \in \mathcal{S}, t \in \mathcal{T}$),$i, j$). This choice of a partition simplifies reasoning about what actions must eventually occur in an execution. However, the necessary liveness properties could also be guaranteed with only two classes: one for $send$(promise($t \in \mathcal{T}$),i, j) actions, using a queue to ensure fairness to each j, and one class for all remaining output actions.

To describe the logically synchronous multicast protocol, we chronicle the events that take

Input Actions:

- $try_i(S)$
  E: $s.try\text{-}set = S$
  $s.need\text{-}set = s'.need\text{-}set \cup S$
  $s.region = \mathrm{T}$
- $rcv(\text{req-promise}, j \in \mathcal{I}, i)$
  E: $s.requests = s'.requests \cup \{j\}$
- $rcv(\text{promise}(t \in \mathcal{T}), j \in \mathcal{I}, i)$
  E: $s.promises\text{-}from[j] = t$
- $multicast\text{-}send_i(m)$
  E: $s.text = m$
  $s.region = \mathrm{M}$
- $rcv(\text{multicast}(m \in \mathcal{S}, t \in \mathcal{T}), j \in \mathcal{I}, i)$
  E: $s.promises\text{-}to[j] = (\infty, n)$
  if $m \neq \epsilon$ **then**
  $s.pending = s'.pending \cup \{(m, t)\}$
- $backout_i$
  E: $s.try\text{-}set = \emptyset$
  $s.region = \mathrm{M}$
- $rcv(\text{adv-promise}(t \in \mathcal{T}), j \in \mathcal{I}, i)$
  E: $s.promises\text{-}to[j] = t$

Output Actions:

- $send(\text{req-promise}, i, j \in \mathcal{I})$
  P: $s'.region \in \{\mathrm{T}, \mathrm{W}\}$
  $j \in s'.need\text{-}set \setminus s'.requested$
  E: $s.requested = s'.requested \cup \{j\}$
- $send(\text{promise}(t \in \mathcal{T}), i, j \in \mathcal{I})$
  P: $j \in s'.requests$
  $t > \max(s'.try\text{-}time, \max(s'.pending).\text{time})$
  E: $s.requests = s'.requests \setminus \{j\}$
  $s.promises\text{-}to[j] = t$
- $ready_i$
  P: $s'.region = \mathrm{T}$
  $s'.pending = \emptyset$
  $\min(s'.promises\text{-}to) \geq s'.try\text{-}time$
  $\forall j \in s'.try\text{-}set,$
  $s'.promises\text{-}from[j] < (\infty, n)$
  E: $s.mctime = s'.try\text{-}time$
  $s.region = \mathrm{R}$
- $send(\text{multicast}(m \in \mathcal{S}, t \in \mathcal{T}), i, j \in \mathcal{I})$
  P: $s'.region = \mathrm{M}$
  $s'.promises\text{-}from[j] < (\infty, n)$
  $t = s'.mctime$
  if $(j \in s'.try\text{-}set)$ **then**
  $m = s'.text$
  else $m = \epsilon$
  E: $s.requested = s'.requested \setminus \{j\}$
  $s.promises\text{-}from[j] = (\infty, n)$.
- $multicast\text{-}rcv_i(m)$
  P: $s'.region \in \{\mathrm{P}, \mathrm{T}\}$
  $(m, t) = \min(s'.pending)$
  $t < \min(s'.promises\text{-}to)$
  E: $s.pending = s'.pending \setminus \{(m, t)\}$
  $s.clock = t$
  if $s'.region = \mathrm{T}$ **then** $s.region = \mathrm{W}$
- $done_i$
  P: $s'.region = \mathrm{M}$
  $s'.requested = \emptyset$
  E: $s.need\text{-}set = \emptyset$
  $s.region = \mathrm{P}$
- $send(\text{adv-promise}(t \in \mathcal{T}), i, j \in \mathcal{I})$
  P: $s'.region \in \{\mathrm{T}, \mathrm{W}\}$
  $\forall k \in s'.try\text{-}set,$
  $s'.promises\text{-}from[k] < (\infty, n)$
  $s'.promises\text{-}from[j] < s'.try\text{-}time$
  $t = s'.try\text{-}time$
  E: $s.promises\text{-}from[j] = s'.try\text{-}time$

Figure 9-3: Transition relation for p_i.

place between u_i's multicast request and the completion of the multicast. To more fully understand this description, it is recommended that the reader follow along in the code for p_i given in Figure 9-3. Unless otherwise noted, the word "process" refers to one of the processes p_i, $i \in \mathcal{I}$. Also, we use the words "time" and "logical time" interchangeably.

To initiate the request to perform a multicast, u_i issues a $try_i(S)$ action, where S is the set of indices of user processes that are to receive the multicast.[5] The $try_i(S)$ action causes p_i to remember S as its *try-set*, insert the elements of S into its *need-set*, and enter its trying region (T). In region T, p_i begins to send "req-promise" messages to each member of *need-set*, keeping track, in the component *requested*, of those requests already made in order to avoid sending duplicate requests. Each process p_j receiving a "req-promise" message eventually responds by sending back a "promise" message with an associated logical time t.[6] The promise means that p_j will not perform or deliver any multicasts with a time greater than t until p_i either relinquishes the promise (by sending a "multicast" message to p_j) or advances the promise (by sending an "adv-promise" message with the later time). One may think of a promise as a roadblock that p_j erects in u_j's computation at some future logical time. The process p_j doesn't allow u_j's computation to advance past that time until the roadblock is removed or advanced by p_i.

The time associated with a promise from p_j is chosen by p_j to be greater than the greatest logical time associated with any message in its pending set, and also to be greater than p_j's *try-time*: the pseudo-variable *try-time* for p_j is defined to be the smallest logical time having process-id j such that

$$try\text{-}time \geq \max(clock, mctime, \max(promises\text{-}from)).$$

One may think of *try-time* as a lower bound on the logical time that p_j could assign to its next multicast.

Each process keeps track of both the times for promises it has granted to other processes (in the *promises-to* array) and the times for promises it has received from other processes (in the *promises-from* array). After receiving a promise from each process p_j in its *try-set*, p_i can issue a $ready_i$ action and assign *mctime* to the current value of *try-time*, provided that (1) p_i's

[5] Recall from the definition of U, that $i \in S$.

[6] Note that p_i sends "req-promise" messages to itself in order to simplify the presentation of the algorithm. A simple optimization would be to eliminate these messages, as well as the "promise" messages that p_i sends to itself in response.

pending set is empty, and (2) all promises p_i has granted with times lower than *try-time* have either been relinquished or advanced past *try-time*. The second condition is present to ensure that u_i receives no multicast messages with logical times less than t after p_i decides to send its multicast. Note that once *mctime* is assigned in a *ready*$_i$ action, it remains fixed for all further processing of u_i's current multicast. Specifically, any further change in the *try-time* leaves the *mctime* unaffected.

When a *ready*$_i$ action occurs, u_i can no longer back out from sending a multicast. The *ready*$_i$ action leaves p_i in the ready region (R), where it waits for u_i to respond with a *multicast-send*$_i(m)$ action. When this action occurs, p_i enters the multicast region (M) and records the desired text of the multicast in its *text* component. In region M, p_i sends "multicast" messages to all processes p_j from whom it holds promises. These messages have two purposes. First, they communicate the *text* and *mctime* of the multicast. Second, they relinquish the promises. If p_i holds a promise from p_j, but j is not in *try-set* (we will see shortly how this may happen), the text argument of the multicast message is set to ϵ, indicating that the promise should be relinquished but that no multicast should be delivered to u_j. After p_i has relinquished all the promises it requested, it may issue a *done*$_i$ action and return to its passive region.

When a process p_j receives a multicast(m, t) message from p_i, it notes that its promise to p_i has been relinquished, and, if $m \neq \epsilon$, inserts the pair (m, t) into its pending set. The message m is eventually delivered to u_j in a *multicast-rcv*$_j(m)$ action when t is the lowest time among the times in p_j's pending set and p_j has no outstanding promises with times less than t. These conditions are necessary to ensure that any later (m', t') pair received by p_j will have $t' > t$ so that multicast messages are delivered in logical time order.

So far in this discussion, we have ignored the fact that many multicasts may be proceeding concurrently. Two complications arise as a result of this concurrency. The first relates to the delivery of a multicast message to a user while that user is itself waiting to send a multicast, and the second results from the need to break deadlock situations in the granting of promises. We now consider each of these complications in turn.

If p_i is in region T and issues a *multicast-rcv*$_i(m)$ action, it enters the waiting region (W) where it waits for a response from u_i. Process u_i, on the basis of the new message m, may decide either to continue trying to perform a multicast or to back out. In case of the former,

u_i issues a $try_i(S')$ action, where S' is not necessarily the same as S. (This explains how p_i may hold promises from processes not named in its *try-set*.) This try_i action is treated just as before. If u_i decides to back out, it instead issues a $backout_i$ action, causing p_i's *try-set* to become empty and causing p_i to enter region M, where it proceeds to relinquish its promises as usual.

In the course of concurrent scheduling of multicasts, deadlock situations may arise in the granting of promises. Consider a situation in which p_i and p_j are trying to send multicasts such that each is in the other's *try-set*. Suppose that all promises received by p_i (including the one received from p_j) are less than some promise received by p_j. Then p_i's *try-time* is less than that of p_j. If p_i has granted p_j a promise less than p_i's own *try-time*, then neither can perform a multicast before the other because each must wait for the other to relinquish its promises. Such deadlock situations are avoided by *promise advancement* as follows. Suppose that p_i has received promises from all processes in its *try-set*, but has determined that it is not yet ready to perform a multicast to relinquish those promises. In order not to block unnecessarily the computation of each process p_j from which p_i has received a promise, p_i may send p_j an "adv-promise" message, informing it of p_i's current *try-time*. Upon receiving an "adv-promise" message from p_i, p_j notes that its promise to p_i has been advanced. This may permit p_j to deliver additional multicast messages from its pending set and/or proceed with its own multicast. In the liveness proof, we will show that these "adv-promise" messages are sufficient to guarantee progress.

In studying the algorithm, one will notice a great deal of nondeterminism in the ordering of events. For example, we have not specified the order in which promises are requested from different processes. As a result of this nondeterminism, the correctness proof of the algorithm is more general, covering many possible implementations of the algorithm.

9.3 Proof of Correctness

Let module P be the composition of all automata p_i, $i \in \mathcal{I}$. In this section, we show that module P solves schedule module M, which implies that the logically synchronous multicast protocol is correct. The organization of the correctness proof closely follows the definition of schedule module M. Clearly, $\text{sig}(P) = \text{sig}(M)$. To show that P solves M, we need to show that all fair behaviors of P satisfy the safety conditions (1 and 2) and the liveness condition (3). We prove

these in order. Throughout the proof, we use subscripts to distinguish the state components of the different automata in P. For example, $region_i$ is the *region* variable in the local state of automaton p_i.

9.3.1 Safety Proof

As we have said, the safety proof consists of showing that all executions of P satisfy conditions (1) and (2) of schedule module M. We start by proving condition (1), that P preserves user well-formedness for all $i \in \mathcal{I}$. Following this, we state some properties of well-formed executions that will be used in the proof of condition (2), as well as in the liveness proof. A key part of proving condition (2) is showing the existence of a proper correspondence relation C on the *multicast-send* and *multicast-rcv* events in any execution α of P, and also a proper total order on the *multicast-send* events in α. To accomplish this, we exhibit particular constructions that produce a correspondence relation C_α and an ordering $\prec_\alpha$ for any execution α of P. We then show that $\prec_\alpha$ is indeed a total order and finally that condition (2) is satisfied. We prove the three parts of condition (2) with the help of several intermediate lemmas.

Turning now to the proof of condition (1), we begin by proving the following relationship between the state of p_i and the definition of $r(i, \alpha)$.

Lemma 9.1: Let α be an execution of P such that α is user well-formed for all $i \in \mathcal{I}$, and let s be the final state of α. Then for all $i \in \mathcal{I}$, $s.region_i = r(i, \alpha)$.

Proof: By induction on the length of α.

Base case: If α is of length 1 (just an initial state), then the schedule of $\alpha|U_i$ is the empty sequence. Therefore, $r(i, \alpha) = \mathrm{P}$ for all $i \in \mathcal{I}$ by definition. In every initial state, $region_i = \mathrm{P}$ for all $i \in \mathcal{I}$, so the lemma holds.

Induction: Let $\alpha = \alpha' \pi s$, where the lemma holds for α' ending in state s'. For all i, if $\pi \notin \mathrm{sig}(U_i)$, then $s.region_i = s'.region_i$. This satisfies the lemma, since $r(i, \alpha) = r(i, \alpha')$ by definition if $\pi \notin \mathrm{sig}(U_i)$.

If $\pi \in \mathrm{sig}(U_i)$, then there are six cases: If π is a try_i action, then $s.region_i = \mathrm{T}$ by the effects clause of try_i actions. Since α is user well-formed for i, $r(i, \alpha) \neq X$. Therefore, for try_i to be the final action of α, it must be that $r(i, \alpha') \in \{\mathrm{P}, \mathrm{W}\}$. So, by definition $r(i, \alpha) = \mathrm{T}$, which is $s.\mathrm{region}_i$. The cases $backout_i$ and $multicast\text{-}send_i$ are argued similarly. If π is a

ready$_i$ action, let s' be the state immediately before the action occurs. By the precondition of *ready*$_i$, we know that s'.region$_i$ = T. So by the induction hypothesis, we know that $r(i, \alpha') = \text{T}$. Therefore, $r(i, \alpha) = \text{R}$ by definition. By the effects clause of *ready*$_i$, s.region$_i$ = R, so $s.region_i = r(i, \alpha)$. The cases *multicast-rcv*$_i$ and *done*$_i$ are argued similarly. ■

Using the above lemma, we now prove that module P satisfies condition (1) of schedule module M.

Theorem 9.2: Module P preserves user well-formedness for i, for all $i \in \mathcal{I}$.

Proof: Consider execution $\alpha = \alpha' \pi s$ of P, where α' is user well-formed for all $i \in \mathcal{I}$ and ends in state s'. Since *try*$_i$ is not an output action of P, we need not consider Part 1 of the user well-formedness definition. Similarly, for Part 2 we need only consider cases where π is an output of P. Since α' is user well-formed for all $i \in \mathcal{I}$, we know that region-sequence(i, α') does not contain X. Therefore, for each output action π of P, we simply need to show that $r(i, \alpha) \neq$ X. We rely on the fact, from Lemma 9.1, that $s'.region_i = r(i, \alpha')$. There are three cases for π an output of P.

1. π = *ready*$_i$: This is only enabled if $s'.region_i = \text{T}$, so $r(i, \alpha') = \text{T}$. Therefore, by definition, $r(i, \alpha) = \text{R}$.

2. π = *done*$_i$: This is only enabled if $s'.region_i = \text{M}$, so $r(i, \alpha') = \text{M}$. Therefore, by definition, $r(i, \alpha) = \text{P}$.

3. π =*multicast-rcv*$_i$: This is only enabled if $s'.region_i \in \{\text{P,T}\}$. If $s'.region_i = \text{P}$, then $r(i, \alpha')$ = P, and by definition $r(i, \alpha) = \text{P}$. If $s'.region_i = \text{T}$, then $r(i, \alpha') = \text{T}$, and by definition $r(i, \alpha) = \text{W}$.

In all cases, $r(i, \alpha) \neq X$. ■

We have just shown that module P preserves user well-formedness for all $i \in \mathcal{I}$. Furthermore, since no *rcv* action is an output of P, it is not possible for P to violate network well-formedness. Therefore, in the remaining proofs we can restrict our attention to well-formed executions only. This motivates the following convenient definition. Let α be an execution of P. We say that α is *admissible* iff α is user well-formed for every $i \in \mathcal{I}$ and α is network well-formed. The

following lemma states some properties of admissible executions that will be used throughout the proof.

Lemma 9.3: Let α be an admissible execution of P. For any $i \in \mathcal{I}$, let α' be a subexecution of P between two successive $done_i$ events, (or between the beginning of α and the first $done_i$ event). Then $\forall j \in \mathcal{I}$, if α' contains an event having *any* of the following forms, then it contains *exactly one* event of each form such that they occur in the following order: *send*(req-promise,i,j), *rcv*(req-promise,i,j), *send*(promise(t),j,i), *rcv*(promise(t),j,i), and *send*(multicast(m,t''),i,j), where $m \in \mathcal{S}, t, t'' \in \mathcal{T}$. Furthermore, any events of the form *send*(adv-promise(t'),i,j), $t' \in \mathcal{T}$, occurring in α' must appear between the last two of the above events.

Proof: The proof is by induction, assuming that the conditions hold for i in the prefix of α up to the beginning of α'.

First we show that no two *send*(req-promise,i,j) events can occur in α'. The action π_1 = *send*(req-promise,i,j) is only enabled when $region_i$ = T and $j \notin requested_i$. When the action occurs, it results in $j \in requested_i$. Elements may be deleted from the set $requested_i$ only while $region_i$ = M. Therefore, another action *send*(req-promise,i,j) cannot occur after π_1 until p_i passes through some state in which $region_i$ = M and then reaches a state in which $region_i$ = T. By Lemma 9.1 and the definition of user well-formedness, this cannot happen without an intervening $done_i$.

Next, we show that if π_1 = *send*(req-promise,i,j) occurs in α, then the next $done_i$ event after π_1 must be preceded by π_5 = *send*(multicast(m,t''),i,j). The action π_1 has as an effect that $j \in requested_i$, and $done_i$ has as a precondition that $requested_i$ is empty. Therefore, since π_5 is the only action that can remove j from $requested_i$, it must occur between π_1 and $done_i$.

Now we show that each event in the sequence must occur in order for the next to occur. By the induction hypothesis, all *send*(req-promise,i,j) actions that occur before α' have their corresponding receives occur before α'. Therefore, by network well-formedness, π_2 = *rcv*(req-promise,i,j) cannot occur before π_1, and only one π_2 action occurs. Action π_3 = *send*(promise(t),j,i) is only enabled when $i \in requests_j$, and the event results in i's removal from that set. Since π_2 is the only action that can cause $i \in requests_j$, it must precede π_3. Again, by network well-formedness and the induction hypothesis, we know that π_3 must precede π_4 = *rcv*(promise(t),j,i). The action π_5 = *send*(multicast(m,t''),i,j) has as a precondition

that *promises-from*$_i[j] < (\infty, n)$. Since π_5 has as an effect that *promises-from*$_i[j] = (\infty, n)$, and since π_4 is the only action that can cause *promises-from*$_i[j] < (\infty, n)$, we know by the induction hypothesis that *promises-from*$_i[j] = (\infty, n)$ at the beginning of α'. Therefore, π_4 must precede π_5.

Since *send*(adv-promise(t'),i, j) has as a precondition that *promises-from*$_i[j] < (\infty, n)$, we know that it cannot occur before π_4 or after π_5. ■

In the remainder of the proof, we often use the above lemma to show the existence or nonexistence of particular events in a portion of an execution.

Conditions (2) and (3) of schedule module M refer to the existence of a correspondence relation and a total order. In completing the proof, it is helpful to fix particular constructions for these as follows. Let α be an execution of P. For all $i \in \mathcal{I}$, if π is a *multicast-send*$_i$ action occurring in α and s is the state immediately preceding π, then we define *time*(π, α) to be $s.mctime_i$. Similarly, if π is a *multicast-rcv*$_i$ action occurring in α and s is the state immediately following π, then we define *time*(π, α) to be $s.clock_i$. We fix the correspondence relation C_α as follows: For all $i, j \in \mathcal{I}$ and for all $m \in \mathcal{S}$, events π_1 = *multicast-send*$_i(m)$ and π_2 = *multicast-rcv*$_j(m)$ correspond in α iff *time*(π_1, α) = *time*(π_2, α). We fix $\prec_\alpha$ to be the ordering as follows: For all π_1, π_2 *multicast-send* actions in α, $\pi_1 \prec_\alpha \pi_2$ iff *time*$(\pi_1, \alpha) <$ *time*(π_2, α).

Before proceeding with the three parts of condition (2), we must first show that $\prec_\alpha$ is indeed a total order on the *multicast-send* events. Recall that the construction of $\prec_\alpha$ is based upon assigning logical times to each *multicast-send*$_i$ event according to the value of $mctime_i$ in the preceding state. In the next lemma, we show that the state component $mctime_i$ is nondecreasing.

Lemma 9.4: Let α be an admissible execution of P. Then for all $i \in \mathcal{I}$, if state s' precedes state s in α, then $s'.mctime_i \leq s.mctime_i$.

Proof: The actions *ready*$_i$ are the only actions that modify $mctime_i$. These actions set $s.mctime_i$ to the value of $s'.try\text{-}time_i$, which is no less than $s'.mctime_i$ by definition. ■

With this lemma, we can now show that each multicast is assigned a unique logical time by the protocol.

Lemma 9.5: Let α be an admissible execution of P. Let $\pi = \textit{multicast-send}_i(m)$ and $\pi' = \textit{multicast-send}_j(m')$ be two events in α. Then $\textit{time}(\pi, \alpha) \neq \textit{time}(\pi', \alpha)$.

Proof: There are two cases, depending on whether or not π and π' are outputs of different user processes. If $i \neq j$, then we know trivially that $\textit{time}(\pi, \alpha) \neq \textit{time}(\pi', \alpha)$ because they differ in the process-id. (The state component $\textit{mctime}_i$ is assigned only to values of $\textit{try-time}_i$, whose process-id component is i by definition.)

If $i = j$, then assume, without loss of generality, that π' precedes π in α. From the definition of user well-formedness, we know that at least one $\textit{ready}_i$ action occurs between π' and π. Let s' be the state from which the last such $\textit{ready}_i$ action occurs, and let s be the resulting state. We know from Lemma 9.4 that $s'.\textit{mctime}_i$ is no less than the value of $\textit{mctime}_i$ in the state after π'. Therefore, if we can show that that $s'.\textit{mctime}_i < s.\textit{mctime}_i$, then we will have proven that $\textit{time}(\pi', \alpha) < \textit{time}(\pi, \alpha)$. By the precondition of $\textit{ready}_i$, we know that in state s', p_i must hold a promise from itself for some logical time t. By Lemma 9.3 and user well-formedness for i, we know that p_i's promise to itself is sent (and received) between the last preceding $\textit{done}_i$ action and state s'. Also by user well-formedness, we know that no $\textit{ready}_i$ action occurs between this $\textit{done}_i$ action and state s', so the value of $\textit{mctime}_i$ is constant over that execution interval. Whenever p_i sends a promise, the promise is assigned a time strictly greater than p_i's own *try-time*, which is, by definition, at least as large as its own *mctime*. Therefore, $t > s'.\textit{mctime}$. Since p_i holds a promise for time t in state s', we know that $s'.\textit{try-time}_i \geq t$. Therefore, since the $\textit{ready}_i$ action assigns $\textit{mctime}_i$ to the value of $\textit{try-time}_i$, we know that $s.\textit{mctime}_i > s'.\textit{mctime}_i$.

■

This immediately implies the desired result that the construction of $\prec$ produces a total order on the *multicast-send* events:

Corollary 9.6: Let α be an admissible execution of P. Then $\prec_\alpha$ is a total order on the *multicast-send* events in α.

Proof: Immediate from Lemma 9.5 and the definition of $\prec_\alpha$. ■

Having shown that $\prec_\alpha$ is a total order, we can turn to the main task of proving condition (2) of schedule module M. We begin with condition (2a).

Theorem 9.7: Let α be an admissible execution of P. Then $\mathcal{C}_\alpha$ is a proper correspondence relation for α.

Proof: Let $\pi = \textit{multicast-send}_i(m)$ be an event in α, and let $try_i(S)$ be the last preceding try_i action. By Lemma 9.5, we know that π is assigned a unique logical time t by the protocol. By the definition of p_i and specifically the preconditions of the $send(\text{multicast}(m,t),i,j)$ action, we know that at most one $send(\text{multicast}(m \neq \epsilon,t),i,j)$ action occurs in α for each $j \in S$ (and that none occurs for $j \notin S$). By network well-formedness, we know that at most one $rcv(\text{multicast}(m,t),i,j)$ occurs in α for each of these sends. So (m,t) is added to $pending_j$ at most once in α, for each $j \in S$ (and never for $j \notin S$). Therefore, by the definition of *multicast-rcv*, at most one $\textit{multicast-rcv}_j(m)$ action corresponds to π for each $j \in S$, and no such actions correspond to π for $j \notin S$. This proves that $\mathcal{C}_\alpha$ satisfies properties 1 and 3 of the definition of a proper correspondence relation.

We now show property 2. By the construction of $\mathcal{C}_\alpha$, each each *multicast-rcv* has an associated logical time and corresponds only to those *multicast-send* actions assigned this time. By Lemma 9.5, each *multicast-send* has a unique logical time, so each *multicast-rcv* can correspond to at most one *multicast-send*. It remains to be shown that each *multicast-rcv* has at least one corresponding *multicast-send*. Let s' be the state from which a $\textit{multicast-rcv}_j(m)$ action occurs and let s be the resulting state. Then by the definition of that action, it must be that $(m,t) \in s'.pending_j$ and $s.clock_i = t$. Therefore, a $rcv(\text{multicast}(m,t),i,j)$ must have occurred prior to s'. By network well-formedness, this event must have been preceded by a $send(\text{multicast}(m,t),i,j)$, which could only have been enabled as a result of a $\textit{multicast-send}_i(m)$ action with an assigned logical time of t. This is the desired corresponding action. ■

The next part of the proof is to show that $\prec_\alpha$ is a proper total order for $\mathcal{C}_\alpha$ and α. In order to accomplish this, we first prove a lemma that states some important invariants on the state of P. The fifth invariant, which states that the minimum time in the pending set of a process p_i is always larger than the clock of that process, is a key piece of the safety proof. Informally, it tells us that no multicast message arrives "too late". This is used to prove a second lemma, that the *clock* component of a process is nondecreasing. This will enable us to show the desired property of $\prec_\alpha$.

Lemma 9.8: Let α be an admissible execution of P. Then for all $i, j \in \mathcal{I}$, the following properties hold for all states s in α.

1. $i \in s.requests_j \Rightarrow s.promises\text{-}to_j[i] = (\infty, n)$
2. $s.promises\text{-}to_j[i] \leq s.promises\text{-}from_i[j]$
3. $s.clock_j < s.promises\text{-}to_j[i]$
4. $(s.region_i \in \{\text{R,M}\} \wedge j \in s.try\text{-}set_i \cap s.requested_i) \Rightarrow s.promises\text{-}from_i[j] \leq s.mctime_i$
5. $s.pending_j \neq \emptyset \Rightarrow s.clock_j < min(s.pending_j).\text{time}$

Proof: Each property is proved by a separate induction on the length of α.[7]

Property (1): If s is an initial state, then for all $i, j \in \mathcal{I}$, $i \notin s.requests_j$, so the statement holds vacuously. The only action which can falsify $s.promises\text{-}to_j[i] = (\infty, n)$ is *send*(promise(t),i, j), but this action removes i from $s.requests_j$. The only action which can add i to $requests_j$ is a *rcv*(req-promise,i, j). So, for the induction step, let $\alpha = \alpha'\pi s$, where $\pi =$ *rcv*(req-promise,i, j) and Property (1) holds for α'. Suppose (for contradiction) that $s.promises\text{-}to_j[i] < (\infty, n)$. This can only be true if there exists some π', either a *send*(promise(t),j, i) or a *rcv*(adv-promise(t),i, j), in α' such that no *rcv*(multicast(m, t'),i, j) occurs between π' and π. However, by Lemma 9.3, every *send*(promise(t),j, i) or *send*(adv-promise(t),i, j) must be followed by a *send*(multicast(m, t'),i, j) before the next *send*(req-promise,i, j) occurs. So by network well-formedness, *rcv*(multicast(m, t'),i, j) must occur between π' and π, giving us a contradiction.

Property (2): The base case, α only a start state, holds since $promises\text{-}from_i[j] = promises\text{-}to_j[i] = (\infty, n)$ for all $i, j \in \mathcal{I}$. Let $\alpha = \alpha' s' \pi s$ be an execution of P, where the property holds in state s'. Now, consider those four actions π that can potentially increase $promises\text{-}to_j[i]$ or decrease $promises\text{-}from_i[j]$:

1. If $\pi =$ *send*(promise(t),j, i), then by Property (1) and the preconditions on π, $s'.promises\text{-}to_j[i] = (\infty, n)$. Therefore, $promises\text{-}to_j[i]$ is not increased by π.

[7] This is in contrast to proofs in which the inductive hypothesis includes all of the invariants.

2. If $\pi = rcv(\text{promise}(t),j,i)$, then $s.promises\text{-}from_i[j] = t$. By network well-formedness, $\pi' = send(\text{promise}(t),j,i)$ must occur earlier in α', leaving $promises\text{-}to_j[i] = t$. The only possible events that could occur between π' and π to make $s.promises\text{-}to_j[i] \neq t$ are $rcv(\text{adv-promise}(t'),i,j)$ or $rcv(\text{multicast}(m,t'),i,j)$. By Lemma 9.3, we know that π' must occur before π such that no $send(\text{adv-promise}(t'),i,j)$ or $send(\text{multicast}(m,t'),i,j)$ occurs between π' and π. By the same lemma, we know that a $\pi'' = rcv(\text{req-promise},i,j)$ occurs before π' such that no $send(\text{adv-promise}(t'),i,j)$ or $send(\text{multicast}(m,t'),i,j)$ occurs between π'' and π'. Hence, by network well-formedness, no $rcv(\text{adv-promise}(t'),i,j)$ or $rcv(\text{multicast}(m,t'),i,j)$ occurs between π' and π.

3. If $\pi = rcv(\text{adv-promise}(t'),i,j)$, then $s.promises\text{-}to_j[i] = t'$. By Lemma 9.3 and network well-formedness, the corresponding $send(\text{adv-promise}(t'),i,j)$ must follow a $\pi' = send(\text{promise}(t),j,i)$ such that no $rcv(\text{multicast}(m,t'''),i,j)$ occurs between them. By the preconditions of $send(\text{adv-promise}(t'),i,j)$, $t' > t$, and that action results in $promises\text{-}from_i[j] = t'$. Furthermore, any other $send(\text{adv-promise}(t''),i,j)$ occurring in α' after $send(\text{adv-promise}(t'),i,j)$ must have $t'' > t'$. Therefore, the property holds.

4. If $\pi = rcv(\text{multicast}(m,t),i,j)$, then $s.promises\text{-}to_j[i] = (\infty,n)$. By network well-formedness, π must be preceded by $\pi' = send(\text{multicast}(m,t),i,j)$, resulting in $promises\text{-}from_i[j] = (\infty,n)$. The only action that can decrease $promises\text{-}from_i[j]$ is a $rcv(\text{promise}(t'),j,i)$. But by Lemma 9.3, any $rcv(\text{promise}(t'),j,i)$ occurring between π' and π must be preceded in that interval by a $send(\text{req-promise},i,j)$ and a $rcv(\text{req-promise},i,j)$. But this violates network well-formedness (S4), so no $rcv(\text{promise}(t'),j,i)$ occurs between π' and π. Therefore $s.promises\text{-}from_i[j] = (\infty,n)$.

Property (3): The base case, α a start state, holds since $clock_j = (0,j)$ and $promises\text{-}to_j[i] = (\infty,n)$ for all $i \in \mathcal{I}$. Now, consider those actions that can potentially increase $clock_j$ or decrease $promises\text{-}to_j[i]$. These are $multicast\text{-}rcv_j$, $send(\text{promise}(t),j,i)$, and $rcv(\text{adv-promise}(t),i,j)$. By definition, the action $multicast\text{-}rcv_j$ sets clock to a value t, such that $\forall i \in \mathcal{I}$, $promises\text{-}to_j[i] > t$. The action $send(\text{promise}(t),j,i)$ sets $promises\text{-}to_j[i] = t$ and is enabled only if $t > try\text{-}time_j$, which is at least $clock_j$ by definition. Finally, the action $rcv(\text{adv-promise}(t),i,j)$ sets $promises\text{-}to_j[i] = t$. To show that $t > clock_j$, we note that $send(\text{adv-promise}(t),i,j)$ is enabled at p_i

only if $promises\text{-}from_i[j] < t$. Therefore, by Property (2), $t > promises\text{-}to_j[i]$ when *send*(adv-promise(t),i,j) occurs. And therefore, $t > promises\text{-}to_j[i]$ when *rcv*(adv-promise(t),i,j) occurs, since Lemma 9.3 and network well-formedness tell us that neither a *rcv*(multicast(m,t'),i,j) nor a *send*(promise(t'),j,i) action can occur between *send*(adv-promise(t),i,j) and *rcv*(adv-promise(t),i,j).

Property (4): The base case, α a start state, holds since $region_i$ = P. Let $\alpha = \alpha'\pi s$, where the property holds after α'. There are two cases.

We first consider the case in which p_i enters region R, and subsequently enters region M. In this case, $\pi = ready_i$, so the property holds by the preconditions and effects of $ready_i$. In that action, *try-time* and *mctime* are made equal, and we note that *mctime* remains unchanged until after p_i exits region M. We also observe that by user well-formedness for i, no try_i actions can occur from regions R or M, so $try\text{-}set_i$ is fixed in R and M. By Lemma 9.3, no new promises from members of *try-set* are received by p_i while in R or M, since those promises have already been received (by precondition of $ready_i$). Therefore, to show that the property holds after all extensions of α in which p_i remains in R or M, we need only show that for all $j \in$ *try-set*, if $promises\text{-}from_i[j]$ is increased, then j is removed from *requested* until the next $done_i$. Since *send*(adv-promise(t),i,j) actions are not enabled from M, we need only consider *send*(multicast(m,t),i,j). However, this action removes j from *requested*. Since *send*(req-promise,i,j) is not enabled in M, j cannot be replaced in *requested* before the next $done_i$.

For the second case, p_i does not enter M from region R. In this case, π must be $backout_i$, by user well-formedness for i. Therefore, by the effects clause of that action, $try\text{-}set_i = \emptyset$, so the property holds vacuously until the next $done_i$.

Property (5): Clearly, the property holds in the initial state. Let $\alpha = \alpha' s' \pi s$ be an execution of P, where the property holds in state s'. The only action that can change $clock_j$ is a $multicast\text{-}rcv_j$, which removes the element from $pending_j$ having the lowest time, and sets $clock_j$ to that time. By Lemma 9.5, no two *multicast-send* actions are assigned the same logical time. So by Lemma 9.3, at most one *send*(multicast(m,t),i,j) occurs for a given time t. And by network well-formedness, at most one *rcv*(multicast(m,t),i,j) occurs. Therefore, no two items in $pending_j$ have the same logical time. So by the induction hypothesis, the property holds.

The action π = *rcv*(multicast($m \neq \epsilon,t$),i,j), for some $i \in \mathcal{I}$, is the only action

that can add elements to $pending_j$. Let s'' be the state from which the corresponding $send(\text{multicast}(m,t),i,j)$ occurs. Since $m \neq \epsilon$ implies that $j \in s''.try\text{-}set_i$, we know from Property (4) that $s''.promises\text{-}from_i[j] \leq t = s''.mctime_i$. Therefore, by Property (2), $s''.promises\text{-}to_j[i] \leq t$. By Lemma 9.3 and network well-formedness, we know that no $send(\text{promise}(t'),j,i)$ or $rcv(\text{adv-promise}(t'),i,j)$ action can occur between s'' and s' to could cause $promises\text{-}to_j[i]$ to increase past t. Therefore $s'.promises\text{-}to_j[i] \leq t$. So, by Property (3), $s'.\text{clock}_j < t$. When π occurs, (m,t) is added to $pending_j$, so Property (5) holds in state s. ∎

We now show that the *clock* state component is nondecreasing.

Lemma 9.9: Let α be an admissible execution of P. Then for all $i \in \mathcal{I}$, if state s' precedes state s in α, then $s'.clock_i \leq s.clock_i$.

Proof: Consider the actions $multicast\text{-}rcv_i$, which are the only actions in which $clock_i$ can be modified. Whenever a $multicast\text{-}rcv_i$ action is enabled, $pending_i$ is nonempty. By definition, a $multicast\text{-}rcv_i$ action results in $clock_i$ being set to the minimum logical time in $pending_i$. By Property (5) of Lemma 9.8, $clock_i$ is less than the minimum logical time in $pending_i$, provided $pending_i$ is nonempty. Therefore, whenever $clock_i$ is modified, its value is increased. ∎

We can now prove property (2b) of schedule module M.

Theorem 9.10: Let α be an admissible execution of P. Then $\prec_\alpha$ is a proper total order for C_α and α.

Proof: We need to show that $\forall i,j,k \in \mathcal{I}$ and $\forall m,m' \in \mathcal{S}$, if $\pi = multicast\text{-}send_i(m)$ and $\pi' = multicast\text{-}send_j(m')$ occur in α with corresponding receives $\hat{\pi} = multicast\text{-}rcv_k(m)$ and $\hat{\pi}' = multicast\text{-}rcv_k(m')$, and if $\prec_\alpha$ orders π' before π, then $\hat{\pi}'$ occurs before $\hat{\pi}$.

From Lemma 9.5, we know that π and π' have associated unique logical times. Let these be t and t', respectively. Since $\prec_\alpha$ orders π' before π, we know that that $t > t'$. Furthermore, by the definition of C_α, we know that $clock_k = t$ in the state immediately after $\hat{\pi}$ and that $clock_k = t'$ in the state immediately after $\hat{\pi}'$. Lemma 9.9 tells us that $clock_k$ is nondecreasing. Therefore, $\hat{\pi}'$ must occur before $\hat{\pi}$. ∎

Finally, we prove property (2c) to complete the safety proof.

Theorem 9.11: Let α be an admissible execution of P with correspondence relation C_α. Then $\forall j \in \mathcal{I}$ and $\forall m, m' \in \mathcal{S}$, if $\pi = \textit{multicast-send}_i(m)$ occurs in α, then no $\textit{multicast-rcv}_i(m')$ occurs between π and the corresponding $\hat{\pi} = \textit{multicast-rcv}_i(m)$.

Proof: Consider the state s from which π occurs, let α' be the prefix of α ending in state s, and let $t = s.\textit{mctime}_i \equiv \textit{time}(\pi, \alpha')$. We know, from user well-formedness for i, that $r(i, \alpha') =$ R. Consider the last action $\textit{ready}_i$ occurring in α', and let s' be the resulting state. (We know such an action must occur, since this is the only action that can result in region R.) We know, again by user well-formedness for i, that $\textit{region}_i =$ R at all states between s' and s.

Suppose (for contradiction) that a $\textit{multicast-rcv}_i(m')$ occurs between π and $\hat{\pi}$. From the definition of $\textit{ready}_i$, we know that $s'.\text{pending}_i = \emptyset$. Therefore, the only way for the $\textit{multicast-rcv}_i(m')$ to occur between s' and $\hat{\pi}$ is for a $\textit{rcv}(\text{multicast}(m' \neq \epsilon, t''), j, i)$ with $t'' < t$ to occur first in that interval. By the preconditions of $\textit{ready}_i$, $s'.\textit{promises-to}_i[j] \geq s'.\textit{try-time} = t$, for all $j \in \mathcal{I}$. Furthermore, any later $\textit{send}(\text{promise}(t'), i, j)$ must have $t' > \textit{try-time}_i$, which is always greater than $\textit{mctime}_i$ by definition. From Lemma 9.4, we know that $\textit{mctime}_i$ is nondecreasing, so $\textit{mctime}_i \geq t$ in all states after s'. Therefore, by Properties (2) and (4) of Lemma 9.8, no $\textit{send}(\text{multicast}(m' \neq \epsilon, t''), j, i)$ with $t'' < t$ can occur after s'. (We ignore $\textit{send}(\text{multicast}(\epsilon, t''), j, i)$ actions here because a $\textit{rcv}(\text{multicast}(\epsilon, t), j, i)$ action does not cause an element to be inserted into the *pending* set. So the only way for a $\textit{rcv}(\text{multicast}(m', t''), j, i)$ with $t'' < t$ to occur between s' and $\hat{\pi}$ is for its corresponding send to occur before s'. If this is the case, then by Properties (2) and (4) of Lemma 9.8, $s'.\textit{promises-to}_i[j] \leq t''$. But this violates the precondition for the $\textit{ready}_i$ action that occurs from state s'. ∎

9.3.2 Liveness Proof

The liveness proof consists of showing that executions of P satisfy condition (3) of schedule module M. We prove the two parts of condition (3) in order. Since the protocol is required to make progress only if the user processes and the network satisfy their liveness properties, we will restrict our attention to only those executions in which the environment is live. This motivates the following definition. Let α be a fair execution of P. We say that α is *well-behaved* iff $\alpha|U_i \in \textit{scheds}(U_i)$ for all $i \in \mathcal{I}$ and $\alpha|N \in \textit{scheds}(N)$. Note that every well-behaved execution is an admissible execution, by the definitions of U_i and N, and the fact that P preserves user

well-formedness for all $i \in \mathcal{I}$.

Before proving condition (3a), we prove four intermediate lemmas. The following lemma states that if a promise is requested, then eventually it is granted.

Lemma 9.12: Let α be a well-behaved execution of P. If event $\pi = send(\text{req-promise},i,j)$ occurs in α then a later $rcv(\text{promise}(t),j,i)$ occurs in α for some $t \in \mathcal{T}$.

Proof: By the definition of $scheds(N)$, a $\pi' = rcv(\text{req-promise},i,j)$ occurs in α after π. By the transition relation for p_j, $i \in requests_j$ in the state after π'. Only a $send(\text{promise}(t),j,i)$ action can cause $i \notin requests_j$. Therefore, a $send(\text{promise}(t),j,i)$ action is enabled in all states after π' until one occurs. Since α is a fair execution and $send(\text{promise}(t),j,i)$ actions are in their own class of the partition, such an action eventually occurs. The definition of $scheds(N)$ tells us that a corresponding $rcv(\text{promise}(t),j,i)$ occurs later in α. ■

The following simple lemma states that if a try_i action occurs, then eventually either $need\text{-}set_i$ becomes fixed, or else a later $ready_i$ or $backout_i$ action occurs.

Lemma 9.13: Let α be a well-behaved execution of P, and let α' be a suffix of α beginning with a try_i action, for $i \in \mathcal{I}$. If no $backout_i$ or $ready_i$ action occurs in α' then there exists a state in α' after which $need\text{-}set_i$ is fixed.

Proof: If no $backout_i$ or $ready_i$ action occurs in α', then from the definitions of p_i and user well-formedness we know that no element is deleted from set $need\text{-}set_i$ in α'. Therefore, since $need\text{-}set_i$ can contain at most n elements, we know that there exists a state in α' after which $need\text{-}set_i$ is not changed. ■

The next lemma states that a process can eventually accumulate promises from all processes named in its *need-set*. This fact will be useful in proving Lemma 9.15.

Lemma 9.14: Let α be a well-behaved execution of P, and let α' be a suffix of α beginning with a try_i action. If neither a $backout_i$ nor a $ready_i$ action occurs in α', then there must exist a point in α' after which the following condition holds for all states s: $\forall j \in s.need\text{-}set_i$, $s.promises\text{-}from_i[j] < (\infty, n)$.

Proof: If no $backout_i$ or $ready_i$ action occurs in α' then by user well-formedness for i, $region_i \in \{\text{T,W}\}$ in all states of α'. From Lemma 9.13, there exists a state s' in α' after which

need-set$_i$ is fixed. Let α'' be the suffix of α' beginning with state s'. Then for each state s'' in α'' and for each $j \in s'.$*need-set*, there are two possibilities: either (1) $j \notin s'.$*requested*$_i$, and *send*(req-promise,i,j) is enabled or (2) $j \in s'.$*requested*$_i$ and *send*(req-promise,i,j) occurs before s' (and after the last preceding *done*$_i$, if one occurs). In case (1), we know that a *send*(req-promise,i,j) must eventually occur since α is a fair execution and such actions form their own class of the partition. So, in either case, Lemma 9.12 tells us that a *rcv*(promise(t),j,i) action must occur in α (after the last *done*$_i$ event, if one occurs). So eventually, *promises-from*$_i[j]$ $< (\infty, n)$ for all $j \in$ *need-set*. No action can occur at p_i in region T or W to cause an entry in the *promises-from*$_i$ array to become (∞, n). Thus, we have the desired result. ■

The final intermediate lemma states that if a process is attempting to perform a multicast, then eventually its *try-time* will stop increasing or the process will perform a multicast.

Lemma 9.15: Let α be a well-behaved execution of P, and let α' be a suffix of α beginning with a *try*$_i$ action. If neither a *backout*$_i$ nor a *ready*$_i$ action occurs in α', then there exist a logical time $t \in \mathcal{T}$ and a state s in α' such that *try-time*$_i = t$ in all states after s.

Proof: If no *backout*$_i$ or *ready*$_i$ action occurs in α', then from Lemmas 9.13 and 9.14 we know that there exists a state s in α' after which *need-set*$_i$ is fixed and p_i holds promises from all processes named in *need-set*$_i$. Let $t = s.$*try-time*$_i$. In order to show that *try-time*$_i$ cannot grow past t in α', we need to show that no new promises arrive at p_i, that p_i does not advance any promises past t, and that *clock*$_i$ and *mctime*$_i$ do not increase past t. Clearly, since *need-set*$_i$ is fixed and p_i holds promises from each process named in *need-set*$_i$, no new promises are requested and no new promises arrive. And by definition, p_i never advances a promise beyond its current *try-time*. Since p_i holds a promise from itself (for a time $\leq t$), we know by Property (3) of Lemma 9.8 that *clock*$_i$ cannot grow past t. Finally, since *mctime*$_i$ is only modified by a *ready*$_i$ action, we know that this is fixed as well. ■

The next two theorems correspond to Conditions (3a) and (3b) of schedule module M. In the first, we assume that there exists a set of blocked processes, and derive a contradiction by showing that the process with the lowest *try-time* must eventually make progress. The promise advancement messages are crucial to this result, because it allows the process with the lowest

try-time to discover this fact. From the previous result, we know that only a finite number of these promise advancement messages are sufficient to ensure that progress is made.

Theorem 9.16: Let α be a well-behaved execution of P. If a try_i occurs in α, then either a $backout_i$ or a $ready_i$ occurs later in α.

Proof: Suppose (for contradiction) that there exists a set $\mathcal{J} \subseteq \mathcal{I}$ such that $\forall j \in \mathcal{J}$, a try_j occurs in α and no later $backout_j$ or $ready_j$ occurs in α. From Lemmas 9.14 and 9.15, we know that there exists a suffix α' of α such that for all $j \in \mathcal{J}$,

1. $try\text{-}time_j$ is fixed in α', and

2. for all states of α', p_j holds a promise from every process named in $try\text{-}set_j \subseteq need\text{-}set_j$.

Let $i \in \mathcal{J}$ be the index of the process with the smallest *try-time* in α', and let t be this *try-time*. To derive a contradiction, we wish to show that a $ready_i$ action occurs in α'.

Given the preconditions on $ready_i$, there are only two ways in which the $ready_i$ action could not be enabled: Either (1) $promises\text{-}to_i[j] < try\text{-}time_i$ for some $j \in \mathcal{I}$, or (2) $pending_i$ is not empty. We consider these in order. By the preconditions on granting a promise, any new promises granted by p_i in α' have logical times greater than t, so we need only consider promises granted before α'. Each process $k \in \mathcal{I} \setminus \mathcal{J}$ makes progress (i.e., has a $backout_k$ or $ready_i$ action), and therefore reaches region M, where it eventually relinquishes every promise held. So, any promise that p_i has granted to any process $p_k \in \mathcal{I} \setminus \mathcal{J}$ for a time less than t must eventually be relinquished. We have already said that the remaining processes $p_j \in \mathcal{J}$ hold promises from all processes named in their *try-sets*. Therefore, since α is a fair execution, a $send(\text{adv-promise}(t'), j, i)$ occurs with t' being the logical time at which $try\text{-}time_j$ is fixed. By the definition of N, a corresponding $rcv(\text{adv-promise}(t'), j, i)$ occurs later in α. Since p_i has the smallest *try-time* among processes named in $\mathcal{J}$, we know that $t' > t$ in all cases. Therefore, all promises that p_i has granted to other processes for times less than t are eventually relinquished or advanced past t. So, for all $j \in \mathcal{I}$, $promises\text{-}to_i[j] \geq try\text{-}time_i$. Therefore, by Property (5) of Lemma 9.8, nothing prevents $multicast\text{-}rcv_i$ actions from occurring to empty $pending_i$, since $try\text{-}time_i \geq clock_i$. Thus, since α is a fair execution, $ready_i$ eventually becomes enabled and must eventually occur. ∎

Finally, we show condition (3b), that a multicast message is eventually delivered to all the destination processes.

Theorem 9.17: Let α be a well-behaved execution of P. If a *multicast-send*$_i(m)$ occurs in α and *try*$_i(S)$ is the last preceding *try*$_i$ action in α, then a *multicast-rcv*$_j(m)$ occurs later in α for each $j \in S$.

Proof: If *multicast-send*$_i(m)$ occurs in α, we know that a *ready*$_i$ must precede it, by user well-formedness for i. By the preconditions of *ready*$_i$, for all $j \in$*try-set*$_i = S$, *promises-from*$_i[j] < (\infty, n)$. Therefore, the actions *send*(multicast(m,t),i,j) remain enabled until they occur. And by definition of N, the corresponding *rcv*(multicast(m,t),i,j) actions must eventually occur.

Once a *rcv*(multicast(m,t),i,j) occurs, the only way for the *multicast-rcv*$_j(m)$ to be prevented is for *promises-to*$_j[k]$ to be less than t, for some $k \in \mathcal{I}$. Note that any new promises granted by p_j must be greater than t until *multicast-rcv*$_j(m)$ occurs, since $t \leq \max($*pending*$)$. Therefore, by Theorem 9.16 and the result of the preceding paragraph, all promises granted by p_j for times less than t must eventually be relinquished. At that point, *promises-to*$_j[k] \geq t$, $\forall k \in \mathcal{I}$, so eventually *multicast-rcv*$_j(m)$ occurs. ■

Theorem 9.18: Module P solves schedule module M.

Proof: Follows immediately from Theorems 9.2, 9.7, 9.10, 9.11, 9.16, and 9.17 and the definition of M. ■

9.4 Complexity Analysis

In this section, we analyze the message and time complexities of the multicast protocol. Let system A be the composition of P and any two automata that solve schedule modules U and N. Let α be an execution of system A. We say that α is an *undeviating execution for* i iff every pair of actions *try*$_i(S)$ and *try*$_i(S')$ either have a *done*$_i$ between them or $S = S'$. That is, in an undeviating execution for i, u_i does not "change its mind" about whether to issue a multicast message or to whom the multicast should be sent.

9.4.1 Message Complexity

There are four types of messages sent in the algorithm: req-promise, promise, adv-promise, and multicast messages. If u_i issues $\pi = try_i(S)$ in an execution of system A, then we say that the following messages occur *as a result of* π: any requests by p_i for promises from any $p_j, j \in S$, any promises sent in response to those requests, any promise advancements by p_i to $p_j, j \in S$, and any multicast messages sent from p_i to $p_j, j \in S$. That is, we charge each try_i action with those messages required to complete the corresponding multicast.

Theorem 9.19: Let α be an undeviating execution for i, where $\alpha|U_i$ contains a $\pi = try_i(S)$. Then at most $4|S|$ network messages occur as a result of π.

Proof: By Lemma 9.3, we know that for each $j \in S$, at most one *send*(req-promise,i, j), one *send*(promise(t),j, i) and one *send*(multicast(m, t'),i, j) occur between π and the completion of the multicast. Now we show that at most one *send*(adv-promise(t''),i, j) occurs. Since the execution is undeviating, promises are requested (and received) only from processes named in S. Since no adv-promises are sent until promises are received from all processes named in S, all promises are advanced at most once, to the same logical time. ∎

In executions that do not have the undeviating property, more messages may be required. In the worst case, the *try-set* grows by one with each try_i action until $|S| = n$, the promise granted by the new process each time exceeds the old *try-time* and is received before the next try_i, and all promises are advanced after each promise is received. In this worst-case scenario, the number of req-promise, promise, and multicast messages are the same as above, but the number of adv-promise messages is $O(n^2)$. In situations where this sort of behavior is expected, one might choose another strategy for advancing promises. Alternative methods of promise advancement are outlined in Section 9.4.3.

9.4.2 Time Complexity

To study the time complexity of the algorithm, we need a method for associating real times with points in an execution. If α is an execution, we say that rt is a *real time assignment for* α if rt maps each event π in α to a real time $rt(\pi, \alpha)$ such that (1) the sequence of times is nondecreasing over the entire execution and (2) increases without bound if α is infinite. If α is

an execution, rt is a real time assignment for α, and π' and π are events in α, we say that the *time between* π' *and* π is $|rt(\pi,\alpha) - rt(\pi',\alpha)|$. We define the *state* s α *at real time* r to be the state s as follows: If r is less than the real time of the first event in α, then s is the initial state. If r is greater than the time of the last event in α, then s is the last state of α. Otherwise, s is the state occurring between the two events π' and π in α such that $rt(\pi',\alpha) < r < rt(\pi,\alpha)$. A more general approach for adding real time to the I/O automaton model is presented in [50], but the above definitions will be sufficient here.

In order to derive meaningful time bounds for the algorithm, we need to make stronger assumptions about message delivery than the eventuality conditions used for the liveness proofs. Therefore, we let d be an upper bound on the time between a *send* event and the corresponding *rcv* (i.e., the message delay). We assume that process step time is insignificant in comparison to d, so we do not impose any lower bound on the time between two successive steps of the algorithm. In fact, to simplify the analysis, we require that if an output action of P is enabled in state s at time r, then either that action occurs at time r, or that action becomes disabled by some other action occurring at time r. Informally, this says that the only delays are in the message system; all processing of a message occurs instantaneously with the receipt of that message. For example, no time elapses between receiving a request for a promise and sending out the promise. We also require that each user respond to *multicast-rcv* and *ready* actions immediately. That is, if a *multicast-rcv*$_i$ action occurs at real time r, then the resulting *try*$_i$ or *backout*$_i$ action occurs at real time r. Similarly, if a *ready*$_i$ action occurs at real time r, then the resulting *multicast-send*$_i$ occurs at real time r. We will restrict our attention to executions of A with real time assignments satisfying the above properties.

We wish to derive an upper bound on the time between making a request to perform a multicast (a *try*$_i$ action) and getting permission to perform the multicast (a *ready*$_i$ action). To accomplish this, we first compute an upper bound on the time for the process with the lowest *try-time* to be able to perform a multicast once it has received all the necessary promises.

Lemma 9.20: Let α be an undeviating execution for i with real time assignment rt. Let s be a state in α such that

1. for all $j \in s.tryset_i$, $s.promises\text{-}from_i[j] < (\infty, n)$, and

2. for all $j \in \mathcal{I}$ with $s.region_j \in \{\text{T,W}\}$, $s.try\text{-}time_i \leq s.try\text{-}time_j$.

If r is the real time of state s, then there exists an event $\pi = ready_i$ in α such that $r < rt(\pi, \alpha) < r + 3d$.

Proof: For all $j \in \mathcal{I}$, if $s.region_j \in \{\text{P,R,B}\}$ and $s.promises\text{-}to_i[j] < (\infty, n)$, then by time $r + d$, a $rcv(\text{multicast}(m,t),j,i)$ action occurs for some $m \in \mathcal{S}$ and $t \in \mathcal{T}$. Furthermore, for all $j \in \mathcal{I}$, if $s.region_j \in \{\text{T,W}\}$ and $s.promises\text{-}to_i[j] < (\infty, n)$, then a $rcv(\text{adv-promise}(t'),j,i)$ action with $t' > s.try\text{-}time_i$ occurs by time $r + 3d$ (one delay for p_j's promise requests, one delay for the promise messages, and one delay for the adv-promise message). Any promise granted by p_i after state s must have a time greater than $s.try\text{-}time_i$, since no action can occur from region T or W to decrease the value of $try\text{-}time_i$. Therefore, by time $r + 3d$, it is the case that $\min(promises\text{-}to_i) > s.try\text{-}time_i$. So, all the multicast messages waiting in $pending_i$ are delivered by time $r + 3d$. Thus, the preconditions for $ready_i$ are satisfied by time $r + 3d$ and the action must occur. ∎

Let α be an execution of P. We say that p_i *depends on* p_j in state s of α iff $s.region_i \in \{\text{T,W}\}$, $s.region_j \in \{\text{T,W}\}$, and $s.try\text{-}time_i > s.promises\text{-}to_i[j]$. We say that p_i *indirectly depends on* p_k in state s iff there is a sequence $p_i, p_{j_1}, p_{j_2}, \ldots, p_k$ such that p_i depends on p_{j_1}, p_{j_1} depends on p_{j_2}, etc. One may think of this sequence as a waiting chain, in which each process is waiting to receive a multicast message from the next process in the chain before it may proceed with its own multicast.

The following theorem says that if z is the length of the longest waiting chain originating at p_i in an undeviating execution and p_i holds promises from all members of its *try-set*, then p_i must wait at most $3d(z + 1)$ time units before completing its multicast.

Theorem 9.21: Let α be an undeviating execution for all $i \in \mathcal{I}$. Suppose that at real time r, p_i is in state s such that $s.promises\text{-}from_i[j] < (\infty, n)$ for all $j \in s.try\text{-}set_i$. Let z be the largest number of processes on which p_i indirectly depends between state s and the next $ready_i$. Then a $ready_i$ occurs by time $r + 3d(z + 1)$.

Proof: At most time $2d$ is required from the time a process requests promises until those promises are received. Therefore, if a process p_j depends on process p_k, it must be that p_j receives a promise request from p_k within time $2d$ of the try_j event. (If the promise request

arrived later, then p_j's *try-time* would already be fixed and p_j would grant a promise for a greater time, contradicting the hypothesis that p_j depends on p_k.) So, extending this argument, the *try-times* for all processes in the longest waiting chain originating at p_i must be fixed by real time $r + 2dz$. So, by Lemma 9.20, we know that if p_l is the process in the waiting chain with the least *try-time*, then a $ready_i$ action must occur by time $r + 2dz + 3d$, shortening the length of the waiting chain by one. Similarly, the next process in line must issue its *ready* action within $3d$ time units, and so on. Therefore, a $ready_i$ occurs by time $r + 2dz + 3dz = r + 5dz$.

However, one can improve on this bound by noticing that by the end of the $3d$ maximum time units between the time the last process in the chain obtains all of its promises until its $ready_i$ occurs, all the remaining processes in the chain will have received any adv-promise messages due them. Therefore, each remaining process waits only for the multicast messages from the processes on which it directly depends. These messages require at most d time units each, and there are z of them in the chain. This gives us a time bound of $2dz + 3d + dz = 3d(z + 1)$. ■

It should not be surprising that the time complexity depends heavily upon pattern of the multicast requests, since this is what determines the dependency order. Since z can be at most n, the delay is at most $3d(n + 1)$.

Note that the worst-case time complexity matches one's expectations about what must happen when all n processes attempt to send multicast messages to every process. A simple inductive argument shows that any protocol requires an $\Omega(dn)$ delay in this worst-case scenario: Since all processes send to all other processes, the conditions of the problem require that the protocol enforce a total order on the multicasts. Thus, the process u whose message is the k^{th} message in the total order must wait at least $d(k - 1)$ time before sending its message, or else it could not have received all $k - 1$ messages ordered before it. (This, of course, assumes that all messages take the maximum time d to arrive.)

The worst-case scenario for an execution without the undeviating property is rather complicated. Process p_1, say, grants promises to all the other processes. Then, processes p_2 through p_n each change their minds n times about their *try-sets* before finally performing multicasts in turn while p_1 waits. On receipt of p_n's multicast message, u_1 changes its mind about its *try-set* and issues a new try_i. But before requesting the additional promises, p_1 first grants new promises to all the other processes $p_2, \ldots, p_n$. Then p_1 requests promises from its new *try-set*

and, receiving those promises, advances its *try-time* past all the new promises it has granted. Thus, the same procedure can start over and repeat itself for a total of n times, since u_1 can change its mind at most n times before a $ready_i$ finally occurs. This worst-case scenario results in a delay of $O(n^3 d)$.

One interesting question is whether a deeper understanding of the time complexity of the algorithm could be obtained by stating a measure of the concurrency inherent in the pattern of *try* actions and deriving a time complexity in terms of that measure. That is, one might measure how well the algorithm performs for a given pattern of multicast requests, and compare this to an optimal strategy for handling that particular pattern. Ideally, an algorithm would perform optimally for all possible request patterns. One complication in this sort of analysis is that the behavior of the protocol itself may influence the pattern of requests.

9.4.3 Possible Optimizations

We begin with two simple optimizations. To simplify the presentation of the algorithm, we chose to deliver only one message in a *multicast-rcv*$_i$ action. As a minor modification, one might wish to send a sequence of messages in each action. Also for the sake of exposition, we chose to let p_i send itself messages over the network. A real implementation, however, would not actually send such messages but simply do some local computation.

A more significant modification would involve not waiting for promises requested from processes not in one's try set. That is, $done_i$ would become enabled when after p_i no longer holds any promises, even if p_i has requested a promise that has not yet been received. One way to achieve this would be for p_i to send out "multicast" messages to every process in *requested*, regardless of whether the promise had actually been received. This modification would require some mechanism for dealing with promises that come in late. One might keep track of the number of earlier $done_i$ actions and tag each request with that number; that tag would be appended to the corresponding promise by the granting process. In this way, promises arriving from an earlier multicast attempt could be ignored.

We mentioned earlier that there are other ways in which promise advancement might be handled. For example, one might not wish to wait until promises have been received from all the members in the *try-set* before advancing promises. Alternatively, one might have a process

request promise advancement from those processes blocking its computation. More specifically, the following options are possible.

1. Spontaneous advancement: This method allows p_i to nondeterministically send advancement messages when it notices that it is holding a promise with a time less than its *try-time*.

2. Advancement on demand: If a process p_j is in T with *try-time* $= t$, and has given a promise to p_i for a time t' less than t, then p_j may send p_i a message, asking it to advance the promise. Upon receiving such a message, if p_i has *try-time* $> t'$, then it will send p_j a promise advancement message.

Deadlock avoidance methods similar to these are discussed in [49]. In both cases, there is a trade-off between the message and time complexities: as one becomes more aggressive about advancing promises to reduce time delays, the number of messages increases.

As a final modification, one might allow a process to make strategic promise requests from processes not in its *need-set*. In this way, if u_i changes its mind about its *try-set*, p_i may not need to wait for additional promises. Of course, requesting too many unneeded promises could adversely affect overall performance by needlessly blocking other processes.

9.5 Summary and Discussion

We have defined the logically synchronous multicast problem and presented a solution that takes advantage of the concurrency inherent in the problem. The strong properties of message delivery order imposed by the problem would make a fault-tolerant solution highly attractive for many applications. However, in a completely asynchronous with undetectable process failures, the properties of the message delivery order are strong enough to make a fault-tolerant solution impossible. The proof of this fact is a reduction to distributed consensus using techniques from [29]. Dolev, Dwork, and Stockmeyer show that if processes can broadcast messages such that message delivery at all processes is consistent with some total order on the broadcasts, then it is possible to implement a distributed consensus protocol that tolerates any number of stopping faults [16]. (Each process simply broadcasts its initial value, and the value in the first message received is used as the decision value.) We know that there does not exist a protocol

for distributed consensus that tolerates even one stopping fault [20]. Therefore, it is impossible to construct a fault-tolerant broadcast protocol in which message delivery at all processes is consistent with a single total ordering of the broadcasts. Since the logically synchronous multicast problem requires message delivery to be consistent with a total ordering of the multicasts (plus other conditions), it also does not admit a fault-tolerant solution. However, in spite of this impossibility result, there do exist useful applications of the logically synchronous multicast protocol we have presented. To conclude the chapter, we illustrate an application of this protocol in an area where we need not be concerned with process failure. Specifically, we consider distributed simulation of I/O automata.

The I/O automaton model has proven useful for describing algorithms and proving their correctness (for examples, see [9, 17, 19, 23, 24, 45, 41, 46, 43, 47, 58, 59]). Therefore, we have developed a simulation system based on that model to aid in the design and understanding of distributed algorithms. *Distributing* the simulation, besides being an interesting exercise in itself, can also reduce the simulation time.

Recall from the definition of the I/O automaton model that input actions of automata are always enabled, and that an action shared by a set S of automata is the output of only one automaton and occurs simultaneously at all automata in S. In addition, the actions enabled in a given state of an automaton may, in general, depend upon all previous actions occurring at that automaton. Furthermore, the fairness condition requires that given an automaton $\mathcal{A}$ and an execution α of $\mathcal{A}$, if some class $C \in \text{part}(\mathcal{A})$ has an action enabled in a state s of α, then either no action in C is enabled in some state s' occurring in α after s', or an action from C occurs in α after state s.

We wish to construct a distributed system for simulating fair executions of a given automaton $\mathcal{A}$, where $\mathcal{A}$ has some finite number of components $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$. To simplify the discussion, we shall assume that each component $\mathcal{A}_i$ has exactly one class in its partition. (The generalization allowing each component to have a finite number of classes is straightforward.) To accomplish this, we simply "plug in" a particular transition relation for each user process u_i in system A such that all of its schedules are in $scheds(U_i)$: We assign process u_i to simulate component $\mathcal{A}_i$. When $\mathcal{A}_i$ has an action π enabled, u_i may issue a $try_i(S)$ action, where S is

the set of automata having π as an action.[8] Then, upon receiving a $ready_i$ input, u_i issues a *multicast-send*(π), where π is the action associated with the previous try_i. We permit u_i to issue a $backout_i$ only if no actions are enabled in $\mathcal{A}_i$. The *multicast-rcv*$_i(\pi')$ input actions are used to drive the simulation of $\mathcal{A}_i$. When a *multicast-rcv*$_i(\pi')$ action occurs, process u_i updates its state based on action π' occurring in $\mathcal{A}_i$.

Given the schedule module M defined earlier, one can verify that this distributed simulation satisfies the definitions of the I/O automaton model. As far as each of the components of the simulation can tell, each action π occurring in the simulation happens simultaneously at every component having π in its signature. It is interesting to see how this construction and the liveness condition of the multicast problem work together to satisfy the fairness condition of the I/O automaton model.

Since the logically synchronous multicast protocol is itself described as an I/O automaton, it would be interesting to implement the distributed simulation described above by *simulating* the multicast protocol in Spectrum. In order to accomplish this, the only extension needed in the simulator itself would be the built-in "automata" for handling device I/O that we described in Section 4.5. Specifically, one would build into the system automata that solve schedule module N. That is, one would implement built-in automata with the same external interface as N that would handle all communication services (opening network connections, and sending out and receiving messages over the network) such that all the behaviors of the composition of these built-in automata were in the set of behaviors of N. Then, the actions in the multicast protocol that are shared with the network would simply be shared with these built-in network server automata. The fact that these network servers make in real system calls would have no impact on the multicast protocol itself, and the simulation could run just as described above.

Thus, there are really four logical components in this simulation. First, there is the network algorithm, running as part of the Spectrum implementation. This interacts with the multicast protocol that consists of multiple I/O automata distributed among several physical machines and running in the Spectrum simulator on each machine. Also being simulated in Spectrum is the collection of user automata that interact with the multicast protocol. And finally, there is the algorithm of interest, which is being simulated by the collection of user automata. Of

[8] In a real implementation, one might have the system determine S based on π.

course, if the goal is to achieve optimal performance of distributed simulation, then one might build the entire multicast protocol and its user processes into the system rather than simulating them as I/O automata. Then, direct distributed simulation of the algorithm of interest would be possible, without needing to write special automata to simulate the algorithm as users of the multicast protocol. That is, the information about which automata (in the algorithm of interest) have output steps enabled would be available *within* the simulator for use in the multicast protocol.

Although the problem described in this chapter has an application to the Spectrum Simulation System, we have presented it as a general problem in a modular framework. The problem statement, the algorithm, and the correctness proof are therefore general results, independent of any particular system or application.

Chapter 10

Conclusion

At the beginning of this thesis, we claimed that that formal models are important not only as a means of analyzing and proving the correctness of distributed algorithms, but also as the basis of software tools for designing better algorithms. We presented the Spectrum Simulation System, a new research tool for the design and study of distributed algorithms. Faithful to the I/O automaton model, Spectrum provides the ability to integrate the entire process of distributed algorithm development: specification, design, debugging, analysis, and proof of correctness. All of this is accomplished within a single formal framework that is natural for describing a wide range of distributed algorithms.

In the early chapters of this thesis, we presented the Spectrum programming language, simulator, and user interface some detail. Then, drawing on related work and our experience using the Spectrum system, we conducted an evaluation of Spectrum in terms of the design goals stated in Chapter 1. In the course of our evaluation, we described a number of possible directions for further work. We explored some of these directions in the latter part of the thesis (shared memory, superposition, and distributed simulation), but many others remain. We conclude the thesis by reviewing two of the more important of these directions.

There is a need for software tools that help us to design and test distributed algorithms in parallel with constructing their correctness proofs. The Spectators and INVARIANT clauses of Spectrum are a start in this direction, especially when superposition is used in conjunction with the INVARIANT clause to check invariants on the global state of the system. Adding user-defined scheduling capability to Spectrum's list of options would be of some help in constructing

"strange" algorithm executions for more rigorous testing. But a particularly fruitful line of research might be to study various proof techniques as the basis of algorithm development tools. When algorithms are developed as refinements of more abstract algorithms, it is often convenient to use mapping proofs to show that the refined algorithm simulates the abstract one. (For examples, see [37], [47], and [3].) Algorithm development tools based on these techniques would be quite useful. For example, one might specify a state mapping and simulate both the refined and the abstract algorithm, mechanically checking at each step of the refined algorithm that there is a corresponding sequence of steps of the higher level algorithm consistent with the given state mapping. The interesting question here is how to accommodate multivalued state mappings and nondeterministic algorithms. Another potential line of research would be to integrate a simulation system like Spectrum with a theorem prover like LP [21, 22] or Isabelle [52] by providing mechanical translation between the Spectrum language and the specification language of the theorem prover. In this way, one could write algorithms as I/O automata, debug them with the help of Spectators or other tools based on proof techniques, and then use a theorem prover to generate the correctness proof.

Another important research area is to provide tools that help provide insight into algorithm efficiency. This means understanding not only worst case time and message complexity, but also expected or average case performance. Using spectators, one can statistically analyze the message complexity of an algorithm. And visualization tools provided by Spectrum can help, for example, to see the level of resource contention or network congestion in an execution. However, it is difficult to study time complexity in Spectrum. Several researchers have already extended the I/O automaton model for the study of real-time systems, and have used the extended model to construct timing-based proofs for distributed algorithms [2, 3, 42, 50]. Their work could form the basis of useful extensions to the Spectrum system. Using their approach, timing information in Spectrum would not be manipulated explicitly by the algorithm being simulated, but timing constraints would be associated with automaton classes and handled automatically by the system. This approach would be in keeping with the separation of logical concerns that is seen throughout the Spectrum design.

As the proliferation of distributed systems continues and our demands for performance and reliability increase, we will be need to find new ways to cope with increasingly complicated

distributed algorithms. The more thoroughly we can integrate formal methods and algorithm development tools, the more likely we are to produce distributed algorithms that are both efficient and correct.

Appendix A

Language Syntax

A grammar defining the syntax for a Spectrum types file follows. Items in **boldface** are variable symbols, items in `typewriter font` are terminal symbols, items in *italics* in angle brackets (<>) are name declarations, items in *italics* (without angle brackets) are uses of declared names. Items in square braces ([ ]) are optional. When several of the same item are permitted, ellipses (...) are used. The percent symbol (%) is the comment marker; the parser ignores all characters from a % to the end of that line. Comments about the grammar rules are surrounded by asterisks (*).

types file ::= [typedef...] actiondef... autdef...

type ::= *type-name* |
`boolean` |
`integer` |
`automaton_id` |
`string` |
`tuple (field,...)` |
`set (type)` |
`multiset (type)` |
`mapping (type, type)` |
`sequence (type)`

field ::= <*field-name*>: *type*

typedef ::= DATA <*type-name*> **type**

* recursive type defs not allowed *

actiondef ::= ACTION <*action-name*> **type**

autdef ::= AUTOMATON <*aut-name*> STATE **type inv maint** [**input**...] [**class**...]

inv ::= INVARIANT **expression**...

* each expression returns a boolean, modifies nothing, doesn't read action arguments *

maint ::= MAINTAIN **expression**...

* each expression modifies only state, doesn't read action arguments *

input ::= INPUT *action-name* [**where**] **effect**

class ::= CLASS [**param**...] [WEIGHT **intvalue**] **output**...

param ::= <*class-param*>:**expression**

* expression refers only to configuration data, param-name takes on the type of the expression *

output ::= OUTPUT *action-name* [**where**] [**pre**] [**select**] [**effect**]

pre ::= PRE **expression**...

* each expression returns a boolean, modifies nothing, doesn't read action arguments *

select ::= SEL **expression**...

* each expression modifies only action arguments, reads anything *

effect ::= EFF **expression**...

* each expression modifies only state, reads anything *

where ::= WHERE **expression**...

* expressions are of type boolean, modify nothing, do not refer to state *

expression ::= **constant** |
variable |
function([expression,...]) |
{ **expression**... } |
expression.*field-name*

* for the last option, expression must be a tuple and field-name must be in that tuple *

variable ::= s | a | c

* state, action argument, or class parameter *

component ::= .*field-name*

* must be a field in the tuple *

constant ::= **intvalue** |
"alphanumeric..." |
boolvalue

alphanumeric ::= **letter** | **numeral**

intvalue ::= [-]**numeral...**

letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

numeral ::= 0|1|2|3|4|5|6|7|8|9

boolean ::= `true` | `false`

function ::= any function listed in Appendix B

Appendix B

Functions

The types of arguments and return values for each function are shown. T, T1, and T2 may be arbitrary types. A function is does not return a result or modify an argument unless it is explicitly stated otherwise.

B.1 Generic

The following functions apply to all data types.

- eq(a:T, b:T) returns BOOLEAN
  returns TRUE if a = b, FALSE otherwise

- less(a:T, b:T) returns BOOLEAN
  returns TRUE if a < b, FALSE otherwise

- less_eq(a:T, b:T) returns BOOLEAN
  returns TRUE if a <= b, FALSE otherwise

- greater(a:T, b:T) returns BOOLEAN
  returns TRUE if a > b, FALSE otherwise

- greater_eq(a:T, b:T) returns BOOLEAN
  returns TRUE if a >= b, FALSE otherwise

- assign(a:T, b:T) modifies a
  copies the value of b into variable a

B.2 Integers

- int_neg(a:INTEGER) returns INTEGER
  returns $-a$
- int_plus(a:INTEGER, b:INTEGER) returns INTEGER
  returns $a+b$
- int_minus(a:INTEGER, b:INTEGER) returns INTEGER
  returns $a-b$
- int_times(a:INTEGER, b:INTEGER) returns INTEGER
  returns $a \times b$
- int_div(a:INTEGER, b:INTEGER) returns INTEGER
  returns $a \div b$, truncated if not evenly divisible
- int_mod(a:INTEGER, b:INTEGER) returns INTEGER
  returns a mod b
- int_random(a:INTEGER, b:INTEGER) returns INTEGER
  returns a random integer in the range [a,b)

B.3 Booleans

- bool_not(a:BOOLEAN) returns BOOLEAN
  returns $\neg a$
- bool_implies(a:BOOLEAN, b:BOOLEAN) returns BOOLEAN
  returns $a \rightarrow b$
- bool_and(a:BOOLEAN, b:BOOLEAN) returns BOOLEAN
  returns $a \wedge b$
- bool_or(a:BOOLEAN, b:BOOLEAN) returns BOOLEAN
  returns $a \vee b$
- bool_xor(a:BOOLEAN, b:BOOLEAN) returns BOOLEAN
  returns $a \oplus b$

B.4 Strings

- str_to_int(a:STRING) returns INTEGER
  converts the (base 10) digits of a into an integer
  if a doesn't begin with digits, 0 is returned

B.5 Configuration Data

- self() return AUTOMATON_ID
  returns the automaton's unique id assigned by the system
- name(a:AUTOMATON_ID) returns STRING
  returns the name assigned to the automaton with id = a
- parent(a:AUTOMATON_ID) returns AUTOMATON_ID
  returns the id of a's direct parent in the composition heirarchy
- in(a:AUTOMATON_ID) returns SET(AUTOMATON_ID)
  the return set contains b iff there is an edge from b to a
- out(a:AUTOMATON_ID) returns SET(AUTOMATON_ID)
  the return set contains b iff there is an edge from a to b
- neighbors(a:AUTOMATON_ID) returns SET(AUTOMATON_ID)
  the return set contains b iff b shares an edge with a
- all_of_type(a:STRING) returns SET(AUTOMATON_ID)
  returns the ids of all instances of the automaton type named a
- edge_rev(a:AUTOMATON_ID, b:AUTOMATON_ID) modifies configuration
  reverses the direction of the edge from a to b
- edge_val(a:AUTOMATON_ID, b:AUTOMATON_ID, c:INTEGER) modifies configuration
  makes c the "value" (color) of the edge from a to b

In the current implementation of Spectrum, the configuration is intended to be *static*. The last two functions above are provided for visualization purposes only, and should be used with caution. For example, if edge_rev is used in program, then the signatures of the automata should not depend on the direction of their adjacent edges; otherwise, the signatures would change during the course of the execution. Also, users should be sure that the named edges actually exist in the configuration. One should especially avoid having multiple automata modify a given edge during the same step. Since the configuration is considered static, the simulator does not restore the configuration on starting a new simulation nor after undoing

a portion of an execution. An interesting possibility for further work would be to extend the system to support a dynamic configuration, including the dynamic creation (and destruction) automaton instances.

B.6 Sets

- set_init(a:SET) modifies a
  initializes a to the empty set
- set_single(a:T) returns SET(T)
  returns the set {a}
- set_empty(a:SET) returns BOOLEAN
  returns TRUE iff the set a is empty
- set_size(a:SET) returns INTEGER
  returns the number of elements in a
- set_el(a:SET(T), b:T) returns BOOLEAN
  returns TRUE iff b$\in$a
- set_insert(a:SET(T), b:T) modifies a
  inserts b into set a
- set_delete(a:SET(T), b:T) modifies a
  deletes b from set a
- set_minimum(a:SET(T)) returns T
  returns the smallest element of a
- set_maximum(a:SET(T)) returns T
  returns the largest element of a
- set_random(a:SET(T)) returns T
  returns a random element of a
- set_union(a:SET(T), b:SET(T)) returns SET(T)
  returns a$\cup$b
- set_diff(a:SET(T), b:SET(T)) returns SET(T)
  returns a$\setminus$b
- set_int(a:SET(T), b:SET(T)) returns SET(T)
  returns a$\cap$b
- set_forall(x,a:SET(T),expression:BOOLEAN) returns BOOLEAN
  x is automatically declared to be of type T
  the expression may involve x
  returns TRUE iff the expression is TRUE for all elements x of a

- set_exists(x,a:SET(T),expression:BOOLEAN) returns BOOLEAN
  x is automatically declared to be of type T
  the expression may involve x
  returns TRUE iff the expression is TRUE for some element x of a

- set_doall(x,a:SET(T),expression) returns BOOLEAN
  x is automatically declared to be of type T
  the expression may involve x
  executes the expression for each element of a

- set_find(x,a:SET(T),expression:BOOLEAN) returns T
  x is automatically declared to be of type T
  the expression may involve x
  returns an element of a that makes the expression TRUE

- set_findall(x,a:SET(T),expression:BOOLEAN) returns SET(T)
  x is automatically declared to be of type T
  the expression may involve x
  returns the set of all elements of a that make the expression TRUE

B.7 Multisets

- mset_init(a:MULTISET) modifies a
  initializes a to the empty multiset

- mset_single(a:T) returns MUTLISET(T)
  returns the multiset {a}

- mset_empty(a:MULTISET) returns BOOLEAN
  returns TRUE iff the multiset a is empty

- mset_size(a:MULTISET) returns INTEGER
  returns the number of elements in a

- mset_el(a:MULTISET(T), b:T) returns BOOLEAN
  returns TRUE iff b$\in$a

- mset_insert(a:MULTISET(T), b:T) modifies a
  inserts b into multiset a

- mset_delete(a:MULTISET(T), b:T) modifies a
  deletes one occurrence of b from multiset a

- mset_minimum(a:MULTISET(T)) returns T
  returns the smallest element of a

- mset_maximum(a:MULTISET(T)) returns T
  returns the largest element of a

- mset_random(a:MULTISET(T)) returns T
  returns a random element from the multiset a
- mset_union(a:MULTISET(T), b:MULTISET(T)) returns MULTISET(T)
  returns a∪b (multiset union)
- mset_diff(a:MULTISET(T), b:MULTISET(T)) returns MULTISET(T)
  returns a\b (multiset difference)
- mset_int(a:MULTISET(T), b:MULTISET(T)) returns MULTISET(T)
  returns a∩b (multiset intersection)

B.8 Sequences

- seq_init(a:SEQUENCE) modifies a
  initializes a to the empty sequence
- seq_empty(a:SEQUENCE) returns BOOLEAN
  returns TRUE iff the sequence a is empty
- seq_size(a:SEQUENCE) returns INTEGER
  returns the length of the sequence a
- seq_el(a:SEQUENCE(T), b:T) returns BOOLEAN
  returns TRUE iff b occurs in a
- seq_back(a:SEQUENCE(T)) returns T
  returns the element at the back of a
- seq_front(a:SEQUENCE(T)) returns T
  returns the element at the front of a
- seq_addb(a:SEQUENCE(T), b:T) modifies a
  inserts b into squence a at the back
- seq_addf(a:SEQUENCE(T), b:T) modifies a
  inserts b into squence a at the front
- seq_del(a:SEQUENCE(T), b:T) modifies a
  deletes some occurrence of b from sequence a
- seq_delb(a:SEQUENCE(T)) returns T modifies a
  deletes and returns the element at the back of a
- seq_delf(a:SEQUENCE(T)) returns T modifies a
  deletes and returns the element at the front of a
- seq_random(a:SEQUENCE(T)) returns T
  returns a random element of a

B.9 Mappings

- map_init(a:MAPPING(T1,T2),b:T2)
  creates a mapping where a(x) = b for all x
- map(a:MAPPING(T1,T2), b:T1, c:T2) modifies a
  makes a(b) = c
- map_eval(a:MAPPING(T1,T2), b:T1) returns T2
  returns a(b)

B.10 Conditionals

- ifthen(expression1:BOOLEAN, expression2)
  if expression1 evaluates to TRUE, expression2 is executed
- ifthenelse(expression1:BOOLEAN, expression2, expression3)
  if expression1 evaluates to TRUE, expression2 is executed
  else expression3 is executed

Bibliography

[1] *Ada Programming Language.* Technical Report ANSI/MIL-STD-1815A-1983, Department of Defense.

[2] 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.

[3] 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. To appear.

[4] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[5] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[6] Rajive Bagrodia. On the design of high performance distributed systems. 1987. Ph.D. dissertation, University of Texas, Austin.

[7] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[8] Andrew Birrell, John Guttag, Jim Horning, and Roy Levin. *Synchronization Primitives for a Multiprocessor: A Formal Specification.* Technical Report 20, Digital Equipment Corporation Stanford Research Center, August 1987.

[9] 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.

[10] Marc H. Brown. *Algorithm Animation.* Technical Report CS–87–05, Ph.D. Thesis, Brown University, Providence, RI, April 1987.

[11] M.H. Brown and R. Sedgewick. A system for algorithm animation. *Computer Graphics*, 18(3):177–186, July 1986.

[12] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[13] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

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

[15] E.W. Dijkstra. Solutions of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[16] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[17] A. Fekete and N. Lynch. The need for headers: an impossibility result for communication over unreliable channels. Also, Technical Memo MIT/LCS/TM-428, MIT Laboratory for Computer Science, May, 1990. To appear in CONCUR. 1990.

[18] A. Fekete, N. Lynch, Y. Mansour, and J Spinelli. *The Data Link Layer: The Impossibility of Implementation in Face of Crashes.* Technical Memo MIT/LCS/TM-355.b, MIT Laboratory for Computer Science, August 1989. Submitted for publication.

[19] 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.

[20] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[21] S. J. Garland and J. V. Guttag. An overview of LP, the Larch prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 137–151, April 1989. Lecture Notes in Computer Science **355**, Springer-Verlag.

[22] S. J. Garland, J. V. Guttag, and J. J. Horning. Debugging Larch shared language specifications. *IEEE Transactions on Software Engineering*, October 1990. To appear.

[23] 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.

[24] 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.

[25] Aparna M. Gupta. I/O automaton based simulation of selected distributed algorithms. June 1990. Senior Thesis.

[26] 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.

[27] David Harel. *On Visual Formalisms*. Technical Report CMU-CS-87-126, Carnegie Mellon Computer Science Department, June 1987.

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

[29] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 276–290, August 1988. A full version is available as MIT Technical Report MIT/LCS/TR-390.

[30] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th Principles of Programming Languages*, pages 13–26, January 1987. Also to appear in *Transactions on Programming Languages and Systems*.

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

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

[33] T.A. Joseph and K.P. Birman. Reliable broadcast protocols. In Mullender, editor, *An Advanced Course on Distributed Computing*, chapter 14, ACM Press, 1989.

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

[35] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690, September 1979.

[36] Leslie Lamport. On interprocess communication. Distributed Computing, 1(1):77–85,86–101, 1986.

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

[38] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, July 1978.

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

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

[41] 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.

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

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

[44] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of a distributed system. *Theoretical Computer Science*, 13:17–43, 1981.

[45] 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.

[46] 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.

[47] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[48] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989.

[49] Jayadev Misra. Distributed discrete-event simulation. *Computing Surveys*, 1(18):39–65, 1986.

[50] Francesmary Modugno, Michael Merritt, and Mark R. Tuttle. Time constrained automata. November 1988. Unpublished manuscript.

[51] Magda F. Nour. *An Automata-Theoretic Model for Unity.* Technical Report MIT/LCS/TM-400, MIT Laboratory for Computer Science, June 1989. Senior Thesis.

[52] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[53] Dick Pountain. *A Tutorial Introduction to Occam Programming.* INMOS, Limited, March 1986.

[54] Gruia-Catalin Roman and Kenneth C. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, 22(10):25–36, October 1989.

[55] Gruia-Catalin Roman, Michael E. Ehlers, H. Conrad Cunningham, and R. Howard Lykins. Toward comprehensive specification of distributed systems. In *In Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 282–289, September 1987.

[56] Robert W. Scheifler and Jim Gettys. *The X Window System.* Technical Report MIT/LCS/TR-368, MIT Laboratory for Computer Science, October 1986.

[57] Fred B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 2(4):179–195, 1982.

[58] Jennifer Welch, Leslie Lamport, and Nancy Lynch. A lattice-structured proof of a minimum spanning tree algorithm. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 28–43, August 1988.

[59] Jennifer L. Welch and Nancy A. Lynch. *Synthesis of Efficient Drinking Philosophers Algorithms.* Technical Report MIT/LCS/TM-417, MIT Laboratory for Computer Science, November 1989. Submitted for publication.

[60] Bernard P. Zeigler. Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation*, 49(5):219–230, 1987.