A Simulator for the IOA Language

by

Anna E. Chefter

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Master of Engineering and Bachelor of Science in Computer Science and Engineering at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

© Massachusetts Institute of Technology 1998. All rights reserved.

Authority:..........................ACHAR

Department of Electrical Engineering and Computer Science May 22, 1998

Certified by..........................Stephen J. Garland
Principal Research Scientist Thesis Supervisor

Certified by..........................Nancy A. Lynch
Cecil H. Green Professor Of Computer Science and Engineering Thesis Supervisor

Accepted by..........................Arthur C. Smith
Chairman, Department Committee on Graduate Students
A Simulator for the IOA Language

by

Anna E. Chefter

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 1998, in partial fulfillment of the requirements for the degrees of Master of Engineering and Bachelor of Science in Computer Science and Engineering

Abstract

With current advances in networking, distributed computing is becoming more commonplace. Distributed systems are hard to design and reason about, because distributed actions can exhibit arbitrary interleaving. In order to make it easier to design and analyze distributed systems, Nancy Lynch and her students have developed a formal mathematical model, the input/output (I/O) automaton model, for describing asynchronous concurrent systems. Based on the I/O automaton model, a new programming language, the IOA language, together with a suite of tools for testing, verifying, and analyzing distributed algorithms is being developed at MIT.

The topic of this thesis is a simulator for the IOA language. Simulation allows one to test and debug algorithms, and it can provide insight that is helpful in understanding algorithms and in constructing correctness proofs for them. The simulator can be used to study the performance of an algorithm under varying conditions. Other contributions of this thesis are the design of an intermediate language that can be used by other IOA tools and the development of a tool that transforms an IOA program into the intermediate representation.

Thesis Supervisor: Stephen J. Garland
Title: Principal Research Scientist

Thesis Supervisor: Nancy A. Lynch
Title: Cecil H. Green Professor Of Computer Science and Engineering
Acknowledgments

I could not have asked for better research advisors than Steve Garland and Nancy Lynch. The IOA system is the result of their vision of the way distributed computing should be done. I would like to thank them for sharing their insights and perspective, and for providing direction and encouragement. Nancy’s rigor and attention to detail and Steve’s desire to spend hours discussing and coding designs with me were invaluable.

I would like to thank my officemate, David Evans, for many useful technical discussions as well as interesting diversions; Ulana Legedza for her helpful suggestions for implementing a simulator; and David Wetherall, a Java guru, who never failed to answer the most obscure questions about Java programming, debugging, and profiling. I would like to thank Professor John Guttag and other members of the SDS group at MIT for making MIT an enjoyable and fun place to do research.

I would like to thank the MIT UROP program for the opportunity to start doing research while still an undergraduate student; I benefitted from the program enormously! The research was also supported by NSF Grant CCR-9504248 for Automated Reasoning in Software Engineering.

A lot of my education at MIT came from being with and keeping up with my friends Pat LoPresti, Kate Dolginova, Mat Hostetter, and Adam Wagman. I would like to thank them for their emotional and technical support throughout the course of my stay at MIT.
## Contents

1 Introduction .......................... 8
   1.1 Thesis Overview .......................... 9
   1.2 Background .......................... 9
   1.3 IOA System .......................... 10
   1.4 Design Goals .......................... 15

2 The Input/Output Automaton Model.................. 18
   2.1 I/O Automata .......................... 19
   2.2 Composition .......................... 20
   2.3 Simulation Relations .......................... 22
   2.4 Alternative Models .......................... 22

3 The IOA Language .......................... 24
   3.1 Overview .......................... 24
   3.2 Structure of IOA Programs .......................... 25
   3.3 Example of a Leader Election Algorithm .......................... 27
   3.4 IOA Terminology .......................... 32

4 Simulator .......................... 35
   4.1 Overview .......................... 36
   4.2 Assumptions .......................... 37
   4.3 Determinators .......................... 39
   4.4 Using the Simulator .......................... 47
## List of Figures

1-1 IOA system architecture ........................................... 12
1-2 IOA design process ................................................ 13
1-3 IOA design using successive refinement ....................... 16

3-1 IOA description of an adder ........................................ 26
3-2 A ring of 4 processes .............................................. 28
3-3 IOA specification of election process ............................ 29
3-4 LSL specification for finite ring of processes .................. 30
3-5 IOA description for a reliable communication channel ....... 30
3-6 IOA specification of the LCR algorithm ......................... 31
3-7 IOA specification of an invariant for the states of automaton LCR . 32
3-8 Dispatching on parameter type .................................... 33

4-1 Eliminating existential quantifier in transition definition ... 40
4-2 Eliminating universal quantifier in the effect clause .......... 41
4-3 Example of nondeterministic choice of initial value for state variable . 42
4-4 Determinator for the Choice automaton .......................... 42
4-5 Determinator for the Adder automaton ............................ 44
4-6 Using for variable in determinator specification .............. 44
4-7 Using RAND in a determinator for Adder ......................... 45
4-8 Closing the Adder automaton using composition ............... 46
4-9 Determinator for AdderClosed automaton ....................... 46
4-10 Numbering several transition definitions ...................... 48
4-11 Using the USER function in a determinator ................... 49
Chapter 1

Introduction

Distributed systems are difficult to design and verify because they must cope with arbitrary interleaving of processor steps. Formal modeling has been applied to reason about distributed algorithms, state problem specifications, describe algorithms precisely, and prove correctness. The input/output automaton model, developed by Nancy Lynch and Mark Tuttle [20], is a formal mathematical model for describing asynchronous concurrent systems. Recently, Stephen Garland and Nancy Lynch have developed a formal language for I/O automata, IOA, which enables the construction of software tools to support the design and analysis of distributed systems.

It is important to have tools for simulating distributed algorithms. First, since formal proofs of correctness are often long, hard, and tedious to construct and read, simulation and testing can help reveal errors in algorithms quickly and easily, before delving into correctness proofs. Second, constructing a correctness proof for an algorithm requires intuition of how the algorithm works, which can be obtained by observing its behavior. Third, a better understanding of an algorithm’s behavior can guide improvement of the algorithm. Fourth, a simulator facilitates the study of the algorithm performance under varying conditions.

In this thesis, we present the design for a simulator for the IOA language and describe its construction and use. Another contribution of this thesis is the development of front end tools for the IOA system.
1.1 Thesis Overview

This thesis is organized as follows. In Chapter 1, we give an introduction to the IOA language and toolset, a system that provides support for mathematics-based distributed programming, and we outline the design goals of the IOA system. In Chapter 2, we review the underlying theoretical model of the IOA system, the input/output automaton model. Chapter 3 introduces the IOA language for describing distributed algorithms as I/O automata and gives examples of IOA specifications of distributed algorithms. The main body of the thesis is Chapter 4, where we describe the usage and give sample runs of the IOA simulator; we end the chapter with a comparison of the simulator to previous research. In Chapter 5, we discuss the implementation of the IOA system and the simulator. Chapter 6 contains a detailed description of a composition transformation, an IOA front-end tool developed by the author. In Chapter 7, we evaluate the design and implementation of the simulator and discuss possibilities for future work.

1.2 Background

The use of computers is undergoing a revolution, leading to widespread use of distributed computing. Until the 1980s, computers were large and expensive, and they operated largely independently of each other. Two major advances in technology began to change that situation. The first was the development of cheap and powerful microprocessors that had computing power comparable to that of mainframe computers. The second was the proliferation of local area networks (LANs), which allowed hundreds of machines to be connected and information to be transferred at rates of 10–100 million bits per second.

The net result of these two technologies is that a large number of CPUs can be connected by a high-speed network to form a distributed system, in contrast with previous centralized systems consisting of a single CPU, memory, and peripherals. Tanenbaum [25] describes the advantages of distributed systems over centralized ones. He
argues that distributed systems are more economical (have higher price/performance ratios), faster (have more total computing power), more manageable (because computing power can be added in small increments), and more reliable (because if one machine crashes the system as a whole can still survive) than centralized systems. Currently, distributed systems are finding applications in such areas as telecommunication, distributed information processing, scientific computing, and real-time process control.

Distributed systems need radically different software than do centralized systems. Unlike sequential algorithms, distributed algorithms must cope with arbitrary interleaving of processor steps. Since a program’s execution can unfold nondeterministically, designing and reasoning about distributed algorithms is inherently difficult. Formal modeling has been applied to reason about distributed algorithms.

The input/output (I/O) automaton model was developed by Nancy Lynch and Mark Tuttle [20] to describe and reason formally about distributed and real-time systems. Professor Lynch’s book *Distributed Algorithms* [19] formulates many algorithms in terms of I/O automata and contains proofs of complexity, reliability, safety and liveness properties of these algorithms. Because of the complicated settings in which distributed algorithms run, the design and verification of these algorithms can be an extremely difficult task. Hence, the model supports a rich set of techniques for proving correctness and other desirable properties of distributed algorithms, including invariant assertion methods, forward and backward simulation methods, and temporal logic methods. Some of the proofs of invariants, simulation relations, and temporal properties have been carried out using computer-aided verification [24].

### 1.3 IOA System

The I/O automaton model provides a solid foundation for the development of distributed algorithms, as shown in [19]. The IOA system is based on the I/O automaton model; the system permits distributed algorithm specification, design, debugging, analysis, and correctness proofs within a single framework. Such integration not only
saves one from translating between different models and languages, but also allows facts discovered during simulation and debugging to be more easily incorporated into correctness proofs, as well as properties used in proofs to be checked mechanically during simulation.

The system provides a language for expressing algorithms as input/output automata, together with a suite of tools that support the production of high-quality distributed software. The IOA language is a programming language for distributed systems that is suitable for both verification and simulation. The toolset provides a variety of validation methods such as theorem proving, model checking, and simulation, which can be used to ensure the correctness of an algorithm. The toolset also supports a development process that starts with a high-level specification, refines that specification by successively adding more details, and finally generates efficient distributed programs, thereby providing a formal connection between verified designs and the corresponding final code.

The high level structure of the IOA system is presented in Figure 1-1. The IOA system is two-tiered. The first tier consists of the IOA language and its front end tools: a parser, a syntactic and semantic checker, prettyprinters, and transformation tools. The second tier consists of the intermediate language and a set of back end tools: a simulator, interfaces to theorem provers and model checkers, and a code generator.

The IOA system contains a variety of analytical tools ranging from light-weight tools, which check the syntax and semantics of an automaton description; to medium-weight tools, which simulate actions of an automaton or provide interfaces to model checkers; and to heavier weight tools, which provide support for proving properties of automata using a theorem prover. Figures 1-2 and 1-3 show how we expect the IOA system to be used in designing distributed algorithms. First, the user designs an algorithm and translates it into IOA specifications. He/she then uses one or more of the verification tools to test, debug, and analyze his/her design. In this process, the user may discover a need to redesign the specifications and properties of the system (see Figure 1-2).
Figure 1-1: IOA system architecture
Develop description of distributed system and its properties

Formalize descriptions in IOA

Use Verification Tools

Generate Code

Figure 1-2: IOA design process
The toolset supports a development process called *successive refinement*. This process is depicted in Figure 1-3. The user first develops an abstract description of a distributed system and its properties. As shown in Figure 1-2, the user formalizes the description in the IOA language, and uses verification tools to check its desired properties. The user then models the architecture of the system using IOA, and proves that this architecture is faithful to the abstract specification by describing a simulation relation (see Section 2.3) between the two levels of description. In the proofs, the user may need to supply information about the correspondence between steps of the high-level and low-level descriptions. The toolset helps users define step correspondences by modifying code or using a special language for defining step correspondences. Continuing in this fashion, the programmer specifies progressively more detailed designs. Once the user reaches low enough level and is confident in the design and correctness of the system, the code generator can be used to translate the IOA specifications into Java or C++ programs.

The IOA system provides three complementary approaches to testing and verifying distributed algorithms:

1. *Simulation*

   The simulator allows the user to observe the run-time behavior of an IOA program; it can be used for debugging and testing the correctness and performance of a distributed algorithm.

2. *Model checking*

   The IOA system will provide an interface to the SPIN [15] model checker. Model checkers provide a different approach to validating a distributed algorithm. A model checker performs an exhaustive test of an automaton’s properties in all reachable states. Model checkers resolve nondeterminism present in IOA specifications by exploring all possible options. Since a model checker automatically explores all reachable states, it is feasible only for systems that do not have many states.
3. **Theorem proving**

An IOA interface to the Larch Prover (LP) [7] can be used to prove properties and invariants of IOA specifications, to prove simulation relations between two specifications, and to prove validity properties for user input and facts about the data types manipulated by the programs. The theorem prover interface submits axioms and goals, translated into the Larch Shared Language (LSL), to LP. Using algebraic substitution and other logical deductions along with user guidance, LP tries to prove that the goals follow from the axioms.

The combination of these three tools with the code generator provides the designer of a distributed system or algorithm with a solid development framework. The toolset is capable of producing efficient distributed programs whose correctness has been fully proved subject to stated assumptions about the environment of the system. The generated code can use specified externally-provided services and underlying hardware. For example, an IOA channel automaton is implemented by externally-provided service such as TCP [21] or MPI [12]. The toolset assumes that built-in and user-supplied sequential code for data type operations is correct; it does not verify this code, though such verification is within the scope of today’s proof technology.

### 1.4 Design Goals

In designing any set of software tools, it is important to formulate and adhere to a set of design principles. The philosophy behind the IOA system is described by the following design principles.

1. The language and tools must be faithful to the formally defined mathematical model. For example, it only makes sense to check simulated executions of an algorithm against the properties asserted by a theorem prover if the semantics of the simulation match the semantics of the proof. Because successful testing alone is not sufficient cause to believe that an algorithm is correct, one should still construct a correctness proof as part of the algorithm development cycle.
Figure 1-3: IOA design using successive refinement
Therefore, it is important that the semantics of the simulation language be consistent with the formal model of the proof.

To meet this goal, as much of the design, verification, and analysis as possible should be done within the IOA language, and therefore in terms of the underlying mathematical model. Only at the latest possible moment should a transition to physical code be made.

2. The language and tools should be natural for expressing a large class of distributed algorithms. However, the main emphasis should be on simplicity and uniformity: complicated language constructs should be avoided in the interest of being able to apply powerful tools to simple statements.

3. The tools should not depend too much on each other; their design spaces should be orthogonal. This goal is essential for the maintenance and scalability of the system. It allows one to use only a subset of the tools, and facilitates incorporating additional testing and verifying tools into the system.

4. The language and tools should encourage experimentation. In general, it should be easy to modify the algorithm being studied and to use the tools. Often a researcher does not know exactly where to look for new insights about an algorithm being developed, but discovers them through exploration and experimentation. It is important that the system facilitate this process. For example, the IOA language should support modular design and decomposition, so that the user can easily change a single component of an algorithm. Furthermore, to facilitate experimentation, the time between modifying an algorithm and being able to use a verifying tool should be short.
Chapter 2

The Input/Output Automaton Model

The Input/Output (I/O) automaton model, developed by Lynch and Tuttle [20], models components of asynchronous concurrent systems as state transition systems. The I/O automaton model is general and simple, and its fundamental notions are mathematical. The execution of an automaton is defined by traces of its external behavior. The model supports techniques for modular design and analysis of distributed systems, including automata composition based on synchronized external actions and description of levels of abstraction based on trace inclusion. The model also supports a rich set of proof methods, including invariant assertions, forward and backward simulation and compositional methods. More details, motivation, examples, and results can be found in [20] and [19].

Lynch’s book Distributed Algorithms [19] describes many algorithms in terms of I/O automata and contains many proofs of various properties of these algorithms and of impossibility results. Careful proofs using the I/O automaton model have been constructed using a variety of techniques for a wide range of algorithms (for more examples, see [18] and [23]). The proofs have been applied to verifying practical distributed systems such as group communication services [6] and distributed shared memory services [5]. (While verifying [5] using the I/O automaton model, a significant error was found and repaired.) In this chapter we present an overview of the I/O
automaton model adapted from Chapter 8 of [19]. In the course of presenting the model we highlight those properties that have been represented in the IOA language and the simulator.

2.1 I/O Automata

I/O automata are best suited for modeling systems whose components operate asynchronously. Each system component is modeled by an I/O automaton, which is a nondeterministic (possibly infinite state) automaton with an action labeling each transition. An automaton’s actions are classified as input, output, or internal. An automaton can restrict 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; closed systems do not interact with their environment.

An automaton’s signature $S$ is a set of actions partitioned into three disjoint sets: the input actions, $\text{in}(S)$, the output actions, $\text{out}(S)$, and the internal actions, $\text{int}(S)$. The external actions, $\text{ext}(S)$, are the input and output actions $\text{in}(S) \cup \text{out}(S)$; the locally controlled actions, $\text{local}(S)$, are $\text{out}(S) \cup \text{int}(S)$. All actions in $S$ are denoted by $\text{acts}(S)$.

Formally, an I/O automaton $A$ consists of five components:

- $\text{sig}(A)$, a signature of $A$
- $\text{states}(A)$, a (possibly infinite) set of states
- $\text{start}(A)$, a nonempty subset of $\text{states}(A)$ known as initial states
- $\text{trans}(A)$, a state-transition relation, such that

$$\text{trans}(A) \subseteq \text{states}(A) \times \text{acts}(\text{sig}(A)) \times \text{states}(A)$$

and for every state $s$ and every input action $\pi$, there is a transition $(s, \pi, s') \in \text{trans}(A)$. Informally, $\text{trans}(A)$ specifies all transitions that can occur for every state and every action.
• \textit{tasks}(A), a task partition, which is an equivalence relation on local(sig(A)).

An element \((s, \pi, s')\) of \textit{trans}(A) is called a transition or step of \(A\). If \((s, \pi, s')\) is a step of \(A\), then \(\pi\) is said to be enabled at state \(s\). Since every input action is enabled at every state, every automaton is input-enabled (i.e., an automaton is unable to block its inputs). The equivalence relation \textit{tasks}(A) is used in the definition of a fair computation. Each class of the partition may be thought of as a separate process.

I/O automata are often described in a precondition-effect style. This style groups together all the transitions \((s, \pi, s')\) that involve each particular type of action into a single piece of code. The code also specifies the preconditions under which the action is permitted to occur, as a predicate on the components of state \(s\). The code specifies the effects that occur as a result of applying \(\pi\) to \(s\). The code in the effects clause and the precondition predicate are executed as one atomic operation. We will say more about I/O code and give examples of I/O automata specifications in the next chapter.

An execution of automaton \(A\) is a finite sequence \(s_0, \pi_1, s_1, ..., \pi_n, s_n\) or an infinite sequence \(s_0, \pi_1, s_1, \pi_2, ...\) of alternating states and actions of \(A\) such that \(s_0 \in \text{start}(A)\) and \((s_i, \pi_i, s_{i+1})\) is a step of \(A\) for every \(i\). A state is said to be reachable in \(A\) if it is the final state of a finite execution of \(A\). The trace of an execution \(\alpha\) of \(A\), denoted \(\text{trace}(\alpha)\), is the subsequence of \(\alpha\) consisting of all its external actions.

\section{2.2 Composition}

Many interesting I/O automata are defined using the composition operation, which can be used to describe an automaton in terms of individual system components. As a special case, a system or an algorithm described by an I/O automaton can be composed with another automaton that represents an I/O automaton model of the system’s environment.

An automaton \(A\) can be defined as a composition of a number of individual automata \(A_1, A_2, ..., A_n\). There are two main requirements on the automata being composed:
1. The set of internal actions of $A_i$ is disjoint from the set of all actions of $A_j$, for all $j \neq i, 1 \leq i, j \leq n$.

2. The sets of output actions $out(A_1), out(A_2), \ldots, out(A_n)$ are mutually disjoint.

We call automata $A_1, A_2, \ldots, A_n$ compatible if their signatures satisfy conditions 1 and 2. The composite automaton $A$ has the signature:

- $int(A) = \bigcup_{1 \leq i \leq n} int(A_i)$
- $out(A) = \bigcup_{1 \leq i \leq n} out(A_i)$
- $in(A) = \bigcup_{1 \leq i \leq n} in(A_i) - out(A)$

The set of states of the composition automaton is the Cartesian product of the sets of states of the component automata.

- $states(A) = \prod_{1 \leq i \leq n} states(A_i)$.

The transitions of a composite automaton are obtained by applying the following rule: if a particular action $\pi$ is in the signature of more than one of the composed automata, then all these automata participate simultaneously in steps involving $\pi$.

- $trans(A)$ is the set of triples $(s, \pi, s')$ such that, for all $i \in \{1, \ldots, n\}$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise, $s_i = s'_i$.

The task partition of the composition is the union of the component task partitions.

- $tasks(A) = \bigcup_{1 \leq i \leq n} tasks(A_i)$.

In [19], Lynch defines a composition operation on a countable collections of automata. We have restated the definition to allow only finite collection of automata to be composed, since this composition operation is the one supported by the IOA tools. We will return to this definition in Chapter 6, where we describe the composition transformation tool of the IOA system front end.
2.3 Simulation Relations

The I/O automaton model supports levels of abstraction based on trace inclusion. High-level descriptions of a distributed system as I/O automata model problem requirements; low-level descriptions are closer to the real implementation of the system. To prove that one automaton implements another, it is enough to define a relationship between the two automata, showing that for any execution of the low-level automaton there is a corresponding execution of the higher-level automaton.

Formally, let $A$ and $B$ be two I/O automata with the same external actions. ($A$ represents the low-level automaton and $B$ represents the high-level automaton.) Suppose $f$ is a binary relation over $\text{states}(A)$ and $\text{states}(B)$. Then $f$ is a simulation relation from $A$ to $B$ provided that the following hold:

1. If $s \in \text{start}(A)$, then $f(s) \cap \text{start}(B) \neq \emptyset$.
2. If $s$ is a reachable state of $A$, $u \in f(s)$ is a reachable state of $B$, and $(s, \pi, s') \in \text{trans}(A)$, then there is an execution fragment $\alpha$ of $B$ starting with $u$ and ending with some $u' \in f(s')$ such that $\text{trace}(\alpha) = \text{trace}(\pi)$.

The first condition requires that any start state of $A$ have some corresponding start state of $B$. The second condition requires that for any step of $A$ and any state of $B$ corresponding to the initial state of the step, there is a corresponding sequence of steps of $B$. In general, this corresponding sequence can consist of none, one, or more steps of $B$ as long as the correspondence between the states and the external behavior of the two automata are preserved.

2.4 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 is contained in [20].

Hoare’s *Communicating Sequential Processes* (CSP) [13] is closely related to the I/O automaton model. A CSP program consists of a set of processes written as
sequential programs. Each program can contain statements that attempt to send or receive data over *channels* connected to other processes. The channels are synchronous, i.e., data transfer occurs simultaneously at both ends of the channel, but only after both the sender and the receiver are at the appropriate points in their programs. Thus, CSP is not suited for describing systems in which the individual processes are autonomous, because, unlike I/O automata, a CSP process that is not prepared to receive data may block a process that is prepared to send data.

Another programming model, UNITY (which stands for Unbounded Nondeterministic Interactive Transformations) [2], uses nondeterministic choice instead of sequential control flow. 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. Statement 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. 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, one declares variables that represent channels and writes statements for sending and receiving data that update those variables. Since there is no notion of “input actions” in UNITY, processes must actively read shared variables in order to become informed of the outputs of other processes. This rules out synchronous interprocess communication. Modularity is a problem in UNITY, since the interfaces between program modules use global variables and are not describable in terms of well-defined sets of actions as in the I/O automaton model.
This chapter introduces the IOA language and gives examples of IOA programs. The IOA language is based on the I/O automaton model discussed in the previous chapter. IOA uses an axiomatic language, the Larch Shared Language (LSL) [14], for defining abstract data types. Part of the discussion of the IOA language is adopted from the IOA manual [9]. For a more thorough presentation of the IOA language, the reader is referred to that document. At the end of this chapter, we discuss a leader election distributed algorithm, describe this algorithm using the I/O automaton model, and give its IOA specification. We will present several simulation runs of this algorithm in the next chapter.

### 3.1 Overview

IOA is a precise language for describing input/output automata and for stating their properties. It was developed by Garland and Lynch as an extension and formalization of the notation used in the Distributed Algorithms book [19] and in [20] and [18]. The language is based on the formal, mathematical I/O automaton state machine model. IOA uses the Larch Shared Language [14] to define the semantics of abstract data types.

Since the I/O automaton model is a reactive system model rather than a sequential programming model, the IOA language cannot simply be a standard sequential pro-
gramming language with some constructs for concurrency and interprocess communication. Instead, the language must be suitable for both verification and simulation. Such a language is hard to design, since for verification an axiomatic language with nondeterministic constructs is preferable, while a deterministic operational language is easier to simulate and translate into real code. In the current design for IOA, data types are defined using the axiomatic Larch Shared Language, while IOA specification of transition definitions include assignment, conditional, iteration, and choose operations.

The language supports two main techniques for building distributed systems out of components. First, the language supports the construction of a system in terms of smaller systems. This technique conforms to the semantics of the composition operation of the I/O automaton model described in Section 2.2. Second, the IOA language supports developing a system design using levels of abstraction. The system is first described at a high level, capturing the essential ideas of an algorithm, and then this specification is successively refined. This technique is based on the formal notion of a simulation relation described in Section 2.3.

3.2 Structure of IOA Programs

In the IOA language, the description of an I/O automaton has four main parts: the action signature, the states, the transitions, and the tasks of the automaton. An automaton’s actions are declared in the signature part. States are described by combinations of values for typed ‘state’ variables. The transitions are given in precondition-effect style (see Section 2.1). Each transition definition has a precondition (pre) which describes a condition on the state that should hold before the transition can be executed, and an effect (eff) which describes how the state changes when the transition is executed. If pre is not specified, then it is assumed to always evaluate to true. The eff part of a transition definition can consist of assignment, conditional, loop, or nondeterministic choice operations. Nondeterminism is useful for generality in high level algorithm design; it is included in the language in the form of choose.
operations. An optional task part describes the task partition. If a task partition is not specified, then it consists of a single task containing all internal and output actions of the automaton.

Figure 3-1 contains a simple IOA description for an automaton, Adder, which gets two integers as input and then outputs their sum. The first line declares the name of the automaton. The signature of the automaton Adder consists of two parameterized actions: add(i, j) and result(k). The types Int and Bool, representing integer and boolean types respectively, are built-in types in IOA.

```plaintext
automaton Adder
signature
  input add(i, j: Int)
  output result(k: Int)
states
  value: Int,
  ready: Bool := false
transitions
  input add(i, j)
  eff value := i + j;
  ready := true
  output result(k)
  pre k = value ∧ ready
  eff ready := false
```

Figure 3-1: IOA description of an adder

It is possible to place constraints on the values of parameters of an action in the signature using the keyword where followed by a predicate. Such constraints restrict the set of actions denoted by the signature. For example, the signature

```plaintext
signature
  input add(i, j: Int) where i > 0 ∧ j > 0
  output result(k: Int) where k > 1
```

could have been used to restrict the values of the input parameters to positive integers and the value of the output parameter to integers greater than 1.

The automaton Adder has two state variables: value, an integer to hold the current sum, and ready, a boolean flag that indicates whether the sum has been
computed. The initial value of value is arbitrary, since it is not set in the description of the state variables; ready is initialized to false.

The input action add(i, j) has no precondition (i.e., its precondition always evaluates to true). Since I/O automata are input-enabled, every input action is always enabled and has no precondition. The effect of transition add(i, j) sets value to the sum of i and j and ready to true. The output action result(k) can occur only when it is enabled, i.e., when ready is set to true and the parameter k is equal to value. Its effect is to reset ready back to false. A trace of Adder is a sequence of external actions such as

\[\text{add}(3, 2), \text{result}(5), \text{add}(1, 2), \text{add}(-1, 1), \text{result}(0), \ldots\]

that starts with an add action, and in which every result action is parameterized by the sum computed by the last add action. Successive result actions must be separated by one or more add actions.

### 3.3 Example of a Leader Election Algorithm

In this section we describe a distributed algorithm that solves the leader election problem in networks with a ring topology. We give an IOA description of the algorithm as a composition of I/O process and channel automata, and we state an invariant for the algorithm.

The problem of electing a unique leader process from among the processes in a network originally arose in the study of local area token ring networks, where a single token circulates around the network, giving its current owner permission to initiate communication. Sometimes, however, the token may be lost, and it becomes necessary to elect the “leader” process that will regenerate the lost token. We consider the LeLann-Chang-Roberts (LCR) leader election algorithm that solves the problem in ring networks. We assume that the network digraph $G$ is a ring consisting of $n$ nodes. Figure 3-2 gives an example of a ring network with 4 processes. The processes associated with the nodes of $G$ do not know their indices, nor those of their
neighbors. Instead, processes are distinguished by a unique identifier (uid). In the LCR algorithm, each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier with its own uid. If the incoming uid is greater than its own, then the process passes the identifier on; if it is equal to its own uid, then the process declares itself a leader. In this algorithm, the process with the largest uid is elected as the leader, since the largest uid is the only one that will pass all the way around the ring.

The IOA description of the process automaton is presented in Figure 3-3. Automaton Process is parameterized by the type I of the process indices and by the process index i. The assumes clause identifies an auxiliary specification, RingIndex (Figure 3-4), that imposes restrictions on the type I. This specification is written in the Larch Shared Language; it requires that there be a ring structure on I induced by operators first, right, and left, and that the operator uid provide a one-to-one mapping from indices of type I to uids of type String. The type declaration on the second line of Figure 3-3 declares Status to be an enumeration of the values waiting, elected, and announced.

The automaton Process(I, i) has two state variables: pending is a multiset of Strings, and status has type Status. Initially, pending is set to \{uid(i)\} and
automaton Process(I: type, i: I)
  assumes RingIndex(I, String)
  type Status = enumeration of waiting, elected, announced
  signature
    input receive(m: String, const left(i), const i)
    output send(m: String, const i, const right(i)),
      leader(m: String, const i)
  states
    pending: Mset[String] := {uid(i)},
    status: Status := waiting
  transitions
    input receive(m, j, i)
      eff if m > uid(i) then pending := insert(m, pending)
      elseif m = uid(i) then status := elected
      fi
    output send(m, i, j)
      pre m ∈ pending
      eff pending := delete(m, pending)
    output leader(m, i)
      pre status = elected ∧ m = uid(i)
      eff status := announced
  tasks
    {send(m, j, right(j)) for m: String, j: I};
    {leader(m, j) for m: String, j: I}  

Figure 3-3: IOA specification of election process

status to waiting. The input action receive(m, left(i), i) compares m, the uid received from the automaton Process(I, left(i)) to the left of Process(I, i) in the ring, with the uid of the automaton itself. If m is greater than the process’s uid, then m is inserted into pending and is sent to the next process in the ring. If m is less than i’s uid, then nothing is done. If m equals to i’s uid, then Process(I, i) is declared the leader. There are two output actions: send(m, i, right(i)), which simply sends a message in pending to the automaton Process(right(i)) on the right in the ring, and leader(m, i), which announces a successful election. The two kinds of output actions are placed in separate tasks, so that a Process automaton whose status is elected must eventually perform a leader action.

Automaton Channel, described in Figure 3-5, represents a reliable communication channel, which neither loses nor reorders messages in transit. The automaton is
RingIndex(I, J): trait
  introduces
  first: \rightarrow I
  left, right: I \rightarrow I
  uid: I \rightarrow J
  asserts with i, j: I
  sort I generated by first, right;
  \exists i \ (\text{right}(i) = \text{first});
  \text{right}(i) = \text{right}(j) \iff i = j;
  \text{left}(\text{right}(i)) = i;
  \text{uid}(i) = \text{uid}(j) \iff i = j

Figure 3-4: LSL specification for finite ring of processes

automaton Channel(M, Index: type, i, j: Index)
  signature
  input send(m: M, const i, const j)
  output receive(m: M, const i, const j)
  states
    buffer: Seq[M] := {}
  transitions
    input send(m, i, j)
      eff buffer := buffer \uplus m
    output receive(m, i, j)
      pre buffer \neq \{\} \land m = \text{head}(buffer)
      eff buffer := \text{tail}(buffer)

Figure 3-5: IOA description for a reliable communication channel
parameterized by a type $M$ of messages, by a type $\text{Index}$ of process indices, and by two indices $i$ and $j$, which represent indices of processes that use the channel. The state of automaton $\text{Channel}$ consists of a sequence of messages, $\text{buffer}$, which is initially empty. The input action $\text{send}(m, i, j)$ has the effect of appending $m$ to $\text{buffer}$. The output action $\text{receive}(m, i, j)$ is enabled when $\text{buffer}$ is not empty and starts with message $m$. The effect of this action is to remove the head element from $\text{buffer}$.

\begin{verbatim}
automaton LCR(I: type)
  assumes RingIndex(I, String)
  compose
    Process(type I, i) for i: I;
    Channel(type String, type I, i, right(i)) for i: I
\end{verbatim}

Figure 3-6: IOA specification of the LCR algorithm

The full LCR leader election algorithm is described in Figure 3-6 as a composition of a set of process automata connected in a ring by reliable communication channels. The \textbf{assumes} statement on the first line repeats the assumption about the type $I$ of process indices in Figure 3-3. The list of automata following the keyword \textbf{compose} describe the composition. This composition consists of one $\text{Process}$ automaton and one $\text{Channel}$ automaton for each element of type $I$. The type parameters $M$ and $\text{Index}$ for the $\text{Channel}$ automata (Figure 3-5) are instantiated by the actual types $\text{String}$ and $I$ of messages and process indices, and the parameters $i$ and $j$ are instantiated by the values $i$ and $\text{right}(i)$, so that each channel connects a process to its right neighbor. In the composition, the input actions $\text{receive}(m, \text{left}(i), i)$ of the automaton $\text{Process}(I, i)$ are identified with the output actions $\text{receive}(m, \text{left}(i), i)$ of the automaton $\text{Channel}(\text{String}, I, \text{left}(i), i)$. Likewise, the input actions $\text{send}(m, i, \text{right}(i))$ of the automaton $\text{Channel}(\text{String}, I, i, \text{right}(i))$ are identified with the output actions of the automaton $\text{Process}(I, i)$. Since all input actions of the channel and process subautomata are identified with output actions of other subautomata, the composite automaton contains only output actions.
invariant of LCR:
\[ \forall i: I \forall j: I \text{ (Process(type I, i).status = elected} \]
\[ \land \text{ Process(type I, j).status = elected } \Rightarrow i = j) \]

Figure 3-7 presents an IOA specification of an invariant for the LCR automaton. The invariant states that at most one process is ever elected as the leader.

3.4 IOA Terminology

In this section we collect some definitions used to describe the parts of IOA specifications of automata.

Definition 1 A primitive IOA automaton description is an IOA program without the compose operator.

An example of a primitive IOA automaton description is the IOA specification for the Adder automaton in Section 3.2.

Definition 2 An action header is an entry, including the where clause, in one of the three (input, output, internal) action lists in the signature of an automaton.

For example, `foo(i: Int, s: String) where P(i, s)` is the action header in the output action list `output foo(i: Int, s: String) where P(i, s)`.

Definition 3 An action can have input, output, or internal action type, depending on whether the action is in the input, output, or internal action list of the automaton’s signature.

Definition 4 An action pattern is a name and a sequence of types for the formal parameters of an action.

For example, `input foo(i: Int, s: String)` has action pattern `foo(Int, String)`.

Definition 5 Two action headers, \(a_1 \in \text{sig}(A_1)\) and \(a_2 \in \text{sig}(A_2)\), match iff their patterns match, i.e., iff \(a_1\) and \(a_2\) have the same names, and the same number, types, and order of the formal parameters.
IOA requires that (see the semantic checks section of the IOA manual [9]):

1. Each automaton has at most one action pattern with a particular name.

2. Each action pattern occurs at most once in each of the input/output/internal action lists of the signature of an automaton. Thus, an action pattern can occur at most three times in a signature definition — once in each action list.

Thus, the following signature is illegal

```plaintext
signature
  input  name(i: Int) where P1(i)
  output name(s: String) where P2(s),
```

because it defines two different action patterns, `name(Int)` and `name(String)`, with the same name.

If the user wants to have a formal parameter that can be either of type `Int` or of type `String`, the user must define the corresponding union type (see Section 9.8 of the IOA manual [9]). Figure 3-8 provides an example of using a union type for `Int` and `String`.

```plaintext
type IntString = union of int: Int, str: String
signature
  output name(x: IntString)
    where if tag(x) = int then P1(x.int) else P2(x.str)
  ...
states
  n: Int
  s: String
transitions
  output name(x: IntString) where tag(x) = int
    pre  x.int = n
    eff ...
  output name(x: IntString) where tag(x) = str
    pre  x.str = s
    eff ...
```

Figure 3-8: Dispatching on parameter type
The same action pattern can occur in different action types, but the user has the proof obligation that the where clauses for these action types do not overlap. For example, the following signature is legal

```
signature
  input name(i: Int) where P(i)
  output name(i: Int) where Q(i),
```

provided the predicate $P(i) \land Q(i)$ is false for every $i$. 
Chapter 4

Simulator

The simulator runs selected executions of an IOA program on a single machine. The simulator checks proposed invariants in the selected executions, generates logs of execution traces, and displays state information upon the user’s request.

In order to simulate an I/O automaton, the user has to resolve all nondeterminism present in its specification. IOA specifications can contain two kinds of nondeterminism: explicit nondeterminism introduced by choose statements and choose parameters, and implicit nondeterminism introduced when more than one transition is enabled at a particular state. The user selects which executions are run by writing determinators for the automaton. The determinator mechanism provides three general resolution techniques for nondeterminism: specifying sets of deterministic alternatives to resolve nondeterministic choices, picking random elements from sets of choices, and prompting the user for input at a point of computation when nondeterminism arises. The simulator provides a choice of scheduling options to resolve implicit nondeterminism, when more than one action is selected for execution.

An important part of the simulator design is the separation of the IOA definition of an automaton and its determinators. As the result of this decision, the user can experiment with a given algorithm in a variety of different simulations.

We start this chapter with an overview of the simulator. In Section 4.2, we state the assumptions on the automata that can be simulated, since the simulator is not capable of simulating an arbitrary IOA program. Section 4.3 discusses the determi-
nator, the mechanism that helps the user of the simulator resolve the nondeterminism of IOA programs, and gives examples of the determinator’s use. Section 4.4 describes the usage of the simulator and gives examples of executions of a simple automaton. In Section 4.5, we restate the LCR algorithm in a form acceptable to the simulator and present several determinators for it. We give two sample executions for the algorithm that were obtained using different scheduling policies and analyze the performance of the simulator. We conclude the chapter by comparing the simulator to previous related projects.

### 4.1 Overview

The biggest problem faced by a simulator of distributed algorithms is resolving nondeterminism. There are several sources of nondeterminism in the IOA language; we distinguish two. **Explicit nondeterminism** arises from choose statements in the effect clauses of transition definitions, choose parameters of transition definitions, choose expressions in the initialization of the state variables, and in non-initialized variables with and without so that constraints. **Implicit nondeterminism** involves the scheduling of enabled actions. If more than one action is enabled in a particular state, then the simulator has to decide which transition, with what parameter values, to schedule for execution.

The input to the simulator consists of the IOA description of an automaton and a determinator for the automaton. The determinator resolves all nondeterminism in the specification for the automaton, including how the automaton’s transitions are scheduled and how values are chosen for choose operators and parameters. A simple language for the determinator provides the user of the simulator with a mechanism for expressing algorithm-specific scheduling rules, randomization over a set of possible parameter values, and halting the simulation for the user's input at a specified point of simulation. The determinator language is described in detail in Section 4.3.

The IOA specification of an automaton together with a determinator for it specifies a set of deterministic executions, which is parameterized by randomization parameters
for the simulation. The IOA definition of an automaton and the determinator for it are written separately; thus, users can experiment with a given algorithm in a variety of different simulations.

At every step of the simulation, the set of possible transitions and their parameter values is determined by the specifications given by the user in the determinator. If more than one transition is specified in a given state, then the scheduler picks one of them according to the scheduling policy selected by the user before the simulation began. The simulator provides the user with several scheduling options.

4.2 Assumptions

The simulator is not capable of simulating an arbitrary IOA program. The following are the restrictions imposed on the IOA programs being simulated (see Section 7.2 for a discussion of removing or relaxing some of these assumptions).

1. IOA Language Restrictions

   - The automaton must be well formed; for example, constraints on transition parameters and state variables should be satisfied, and the components of a composite automaton should be compatible.

2. Simulator Restrictions

   - All quantified variables must be bounded, that is, have a finite set of possible values.
   - The simulator must have an implementation for every operator used in the definition of the automaton. The simulator has a library of implementations for the data types and operators built into the IOA language. If the IOA specification of the automaton uses a non-built-in data type, then the user must provide an implementation for this data type, as described in Section 5.4.2. In this case, the user is responsible for the correctness of the code.
• The number of component automata must be finite. Thus the following is not a valid input to the simulator (but is a valid IOA specification):

\textbf{automaton A}

\texttt{compose B(i) for i: Int.}

3. \textit{Restrictions of the Current Implementation}

• The use of quantified variables in the \textit{IOA} description of the automaton is not allowed. The user must rewrite the program to eliminate quantifiers. For example, some existential quantifiers used in transition definitions can be translated using \texttt{choose} parameters, as shown on Figure 4-1. Some universal quantifiers in the effect clause of transition definitions can be translated using the \texttt{for} operator, as shown on Figure 4-2. In this example, \texttt{P} is a predicate testing an element of enumeration type \texttt{Color}.

• Only variables and simple constants are allowed as parameters to the transition definitions of the automaton. It is possible to translate a transition with an expression as a parameter to one that has only variables as its parameters using a \texttt{choose} parameter. This transformation is described in Section 7.2 in Figure 7-1.

• The parameterized components of composite automata cannot be instantiated with constrained variables. This restriction is imposed because of a difficulty in naming states of the resulting automaton (see Section 6.3). Thus, the simulator does not accept the following examples.

\texttt{automaton A}

\texttt{compose B(i) for i: Int where 1 \leq i \land i \leq 3.}

\texttt{type Color = enumeration of white, red, black}

\texttt{automaton C}

\texttt{compose D(i) for i: Color.}

Instead, these examples must be rewritten as follows:
automaton A
compose B(1); B(2); B(3).

type Color = enumeration of white, red, black
automaton C
compose D(white); D(red); D(black).

- Only one automaton can be simulated at a time. Simulating several automata could be useful when the user wants to test a simulation relation (see Section 2.3). We give suggestions on how to implement this kind of coupled simulation in Section 7.2.

- The user must rewrite the specification of the automaton so that all choose variables have different names. This is necessary because the determinator uses a global naming scheme for all choose variables of an automaton.

The simulator has a predefined initialization value for every built-in type. These values are used for every non-initialized state variable; therefore, the user does not have to resolve the nondeterminism involved in the initialization of state variables. The user can also specify a set of initial values for a state variable, and the simulator will pick one of the specified values at random.

4.3 Determinators

As mentioned in Section 4.1, the simulator needs to resolve all nondeterminism present in IOA specifications. An I/O automaton to be simulated is transformed using a determinator into a particular deterministic version of the automaton.

The language for describing determinators is designed so that the user can assist the simulator in resolving nondeterminism present in the specification of automata. It provides a mechanism for expressing algorithm-specific scheduling rules, such as "if any automaton has more than fifty messages in its buffer, then give it priority to take a step." It also enables users to specify which transition definitions and parameter
values should be chosen at each state. Determinators are written separately from automaton specifications.

A determinator has two parts. The first resolves all choose operators and choose parameters in the automaton’s definition; it is introduced by the keyword choose. The second resolves nondeterminism in action scheduling; it is introduced by the keyword transitions. Appendix B gives a BNF grammar for the determinator language. In the rest of this section we argue for the necessity of the two parts of the determinator language, give examples of each part, and describe their usage.

In the choose section, the user must resolve every choose parameter and every choose operator used in the specification of the automaton by specifying a finite multiset of possibilities. The user must also rewrite the specification of the automaton so that all choose variables have different names. This is necessary because a global naming scheme is used in the choose section of the determinator. When the simulator

---

1The implementation of this part of the determinator is not complete yet.
automaton TestColor
  type Color = enumeration of white, red, black
  signature
    output report(b: Bool)
  states
    done: Bool := false
  transitions
    output report(b)
    pre 'b = done
    eff done := true
    for c: Color in Color do
      done := done \& P(c)
    od

Figure 4-2: Eliminating universal quantifier in the effect clause
encounters a \texttt{choose} variable, it picks a random element in the finite multiset of values for this variable specified in the \texttt{choose} section of the determinator. Currently, the simulator uses a uniform distribution for selecting this random element; if the user wants value $v_1$ to be selected twice as frequently as value $v_2$, then the number of occurrences of $v_1$ in the multiset should be twice that of $v_2$.

Consider the automaton \texttt{Choice} defined in Figure 4-3. The state variable \texttt{num} is initialized nondeterministically to some value of the variable \texttt{n} that satisfies the predicate $1 \leq n \land n \leq 3$, that is, to one of the three values 1, 2, or 3. Figure 4-4 contains a determinator for the automaton \texttt{Choice}. The \texttt{choose} section in the determinator shown in Figure 4-4 instructs the simulator to pick this value in the multiset \{1, 2, 3\}. The \texttt{transitions} section specifies scheduling information and is described later in this section.

\begin{verbatim}
automaton Choice
signature
  output result(i: Int)
states
  num: Int := choose n where 1 \leq n \land n \leq 3,
  done: Bool := false
transitions
  output result(i)
    pre ¬done \land i = num
    eff done := true

Figure 4-3: Example of nondeterministic choice of initial value for state variable
\end{verbatim}

\begin{verbatim}
simulate Choice
  choose n: Int in \{1, 2, 3\}
transitions
  if ¬done then
    result(num)

Figure 4-4: Determinator for the Choice automaton
\end{verbatim}

Any \texttt{so that} or \texttt{where} constraint on the initial state or on the post state of a transition is checked after initialization or execution of the transition, respectively.
For example, the \texttt{where} clause in the \texttt{Choice} automaton in Figure 4-3 is checked after the initialization of its state variables. The user can use the theorem prover to check that the provided choices satisfy any required constraints (expressed by \texttt{where} clauses, preconditions, or \texttt{so that} predicates) before or during the simulation. If any of the constraints are violated, the simulator informs the user and stops the simulation.

The simulator must also resolve implicit nondeterminism. If more than one transition definition is enabled in a particular state, then the simulator must decide which one to execute. Moreover, since an action header may contain action parameters, every parameterized action can be viewed as an additional source of nondeterminism for the scheduler. The user must explicitly resolve nondeterminism by specifying which transition definition should be executed in each state, and with which parameter values. The user writes this information in the \texttt{transitions} part of a determinator.

In the \texttt{transitions} sections, the user provides a list of conditional clauses that specify the set of selected transitions and their parameter values. The form of each clause is either \texttt{if <test> then <set1>} or \texttt{if <test> then <set1> else <set2>}, where \texttt{<test>} is a predicate on state variables of the automaton, and \texttt{<set1>} and \texttt{<set2>} are finite sets of transitions with parameter values for these transitions. If \texttt{<test>} evaluates to true in a particular state, then \texttt{<set1>} is the set of transitions and parameter values for them that are selected for execution at this state, otherwise \texttt{<set2>} is the selected set.

At each step of the simulation, the simulator determines the set of selected transitions by evaluating the list of \texttt{if} statements in the \texttt{transitions} section of the determinator. The set of all selected transitions and parameter values for them at a state is the multiset union of the results of evaluating each \texttt{if} statement at this state.\footnote{It may be a multiset union in which the same transition definition appears more than once with different weights or different time estimates. The weight of the transition definition is the sum of the specified weights, the time estimate for it is the inverse of the sum of the inverses of the specified time estimates.} The scheduler chooses one of the specified transitions according to the scheduling policy selected by the user before the simulation began. If there are no transitions specified for a state, then the simulator informs the user and stops the simulation.
Possible scheduling policies for transitions are discussed in Section 5.4.1.

Consider again the determinator for the automaton Choice given on Figure 4-4. The transitions section of the determinator specifies that if state variable done is false, then the set of selected transitions consists of a single transition result with parameter value num; otherwise, no transitions are selected.

Figure 4-5 gives an example of a determinator for simulating the automaton Adder.

simulate Adder
  transitions
    if ready then
      result(value), add(value, value+1)
    else
      add(1, 2)

Figure 4-5: Determinator for the Adder automaton

(The IOA specification for Adder appears in Figure 3-1.) This determinator specifies that, when ready is true, one of the two transition definitions, result(value) and add(value, value+1), can be scheduled for execution. In all other states, the add(1, 2) transition should be executed.

The expressions that specify parameter values for transitions can use only state variables of the automaton. The user can also introduce a for variable to parameterize the finite set of selected transitions, as illustrated in Figure 4-6. This determinator for automaton Adder specifies that if state variable ready is false, then the set of selected transitions is add(1, 1), add(2, 1), and add(3, 1).

simulate Adder
  transitions
    if ready then
      result(value)
    else
      add(i, 1) for i: Int in {1, 2, 3}

Figure 4-6: Using for variable in determinator specification
The determinator language has two special functions, **RAND** and **USER**. These functions can be applied to the primitive data types (i.e., **Bool**, **Int**, **Nat**, and **Real**) supported by the simulator. They can be used in both the choose and transitions parts of a determinator.

A call to the **RAND** function applied to a primitive built-in type returns a random value of this type. Figure 4-7 demonstrates the use of the **RAND** function in another determinator for Adder.

```plaintext
simulate Adder
transitions
if ready then
    result(value)
else
    add(RAND(Int), RAND(Int))
```

Figure 4-7: Using **RAND** in a determinator for Adder

determinator for Adder. If the state variable **ready** is **true**, then a transition for **result** is simulated with its parameter set to the current value of the state variable **value**. Otherwise, the transition **add** is executed with its parameters set to the values supplied by the **RAND** function.

The use of the **USER** function is similar to that of **RAND**. When a call to the **USER** function applied to a primitive built-in type, **T**, is encountered during a simulation, the user is prompted to enter the value of this type. The simulator parses the input and uses the resulting value of type **T**.

As an alternative to using a determinator, it is possible to resolve some of the implicit nondeterminism by modifying the specifications of the simulated automaton. For example, the user can augment the automaton with new state variables containing scheduling information, can add extra constraints involving the new scheduling variables to the preconditions of transitions, and can add extra statements to the effects of transitions to maintain the scheduling variables. This conversion must be done manually, without the help of the determinator mechanism. The advantage of doing the conversion manually is that the same deterministic IOA specifications can be reused with the theorem prover, the simulator, and the code generator, while the
determinator can be used only with the simulator and the code generator.

Figure 4-8 gives an example of resolving some of the implicit nondeterminism in automaton Adder in this fashion. This example defines an “environment automaton” AdderEnv to supply inputs for the automaton Adder and uses the composition operation to construct a deterministic system to simulate. The automaton AdderEnv uses two state variables, nums and next, to provide parameters for the output action add(i, j). The precondition for the transition definition add(i, j) constrains the values of i and j in the terms of these state variables, and the effect clause causes new values to be used when the action is simulated again.

```plaintext
automaton AdderEnv
signature
  output add(i, j: Int)
states
  nums: Array[Int] := {1, 2, 3, 4},
  next: Int := 1
transitions
  output add(i, j)
  pre i = nums[next] A j = nums[next+1] A next ≤ 4
  eff next := next + 2
```

```plaintext
automaton AdderClosed
compose Adder; AdderEnv
```

Figure 4-8: Closing the Adder automaton using composition

```plaintext
simulate AdderClosed
transitions
  if ready then
    result(value)
  else
    add(nums[next], nums[next+1])
```

Figure 4-9: Determinator for AdderClosed automaton

Figure 4-9 gives an example of a determinator for the composite automaton AdderClosed. This determinator removes the remaining nondeterminism by explicitly specifying a transition definition and its parameters for each state, using a simple
function applied to the new variables. Thus, even when all nondeterminism is re-
moved in an IOA specification, the simulator still needs help from a determinator in
identifying just which transition is enabled in a state and for which parameter values.

The simulator can simulate both input and output actions; thus, the simulated
automaton does not have to be closed (restricted to have only output and internal
actions). The determinator in a way “closes” the automaton, so that all simulated
input actions can be thought of as output actions.

Finally, the determinator language provides support for distinguishing transition
definitions with the same action pattern and action type. The IOA language allows
the user to specify several transition definitions for a particular action pattern and
action type; however, the simulator needs to know which one of them to execute. If
there are several transition definitions for a particular action pattern and action type,
then the intermediate representation of the automaton numbers them in the order of
their appearance (see Section 5.3 for details). In writing a determinator specification
for a transition with multiple transition definitions for its action pattern and action
type, one appends the number of the transition definition (from among those with the
same pattern and type) to the action header of the transition as shown in Figure 4-10.

4.4 Using the Simulator

In this section we describe how the simulator is used and give sample execution runs
of a simple automaton. A more interesting example is given in Section 4.5.

The user invokes the simulator by a command line and then responds to any
prompts for inputs. The simulation is run for a specified number of steps or until no
transitions are selected. If a specified invariant for the automaton or any constraint
on state variables or transition parameters is violated, the simulation is stopped and
the user is informed about the failure. If the determinator for a simulated automaton
contains the USER function, then the simulation will be halted at the point where the
call to the USER function is encountered.
automaton AdderCountZero
signature
    input   add(i, j: Int)
    output  result(k: Int), zero_reported(k: Int)
states
    value: Int,
    ready: Bool := false,
    count: Int := 0
 transitions
    input add(i, j)
      eff value := i + j;
      ready := true
    output result(k)
      pre k = value ∧ ready where k = 0
      eff count := count + 1
      ready := false
    output result(k)
      pre k = value ∧ ready where k ≠ 0
      eff ready := false
    output zero_reported(k: Int)
      pre k = count

simulate AdderCountZero
transitions
    if ready ∧ value = 0 then
      result(value):[1], zero_reported(count)
    if ready ∧ value ≠ 0 then
      result(value):[2], zero_reported(count)
    else
      add(value, value+1), zero_reported(count)

Figure 4-10: Numbering several transition definitions
For example, consider the simulation of the automaton Adder using the determinator given in Figure 4-11. A sample execution run of the automaton Adder with this determinator is given in Figure 4-12. The simulator was invoked using the following command line:

```
java larch.simulator -rand -step 2 -acts 1 < Adder.il
```

The command `java` indicates that the simulator is written in the Java programming language; it invokes the Java interpreter for `larch.simulator`, the simulator program. The file `Adder.il` contains the intermediate representations of the IOA description of automaton Adder and the determinator in Figure 4-11 (see Section 5.3).

```
simulate Adder
transitions
  if ready then
    result(value)
  else
    add(USER(Int), 1)
```

Figure 4-11: Using the USER function in a determinator

Simulation of automaton Adder

states:
  ready = false
  value = 0
actions:
enter the 1st parameter of type Int for transition add
  1
  add(1, 1)
states:
  ready = true
  value = 2
actions: result(2)
states:
  ready = false
  value = 2
simulation stopped: executed 2 transitions

Figure 4-12: A sample execution run with user interaction

There are three optional command flags to the simulator. The first indicates the scheduling policy, the second indicates the number of steps in the execution to
be simulated, and the third indicates the number of steps between two consecutive
reports of state information.

The first command flag of the simulator is used for selecting a scheduling policy. The
scheduler for the automaton is controlled by the determinator and the user-selected
scheduling policy. The user can select one of three scheduling policies by using the
following command line flags: rand for the randomized scheduler, round for the
round robin scheduler, and time for a scheduler with time estimates. The randomized
scheduler with equal weights is used by default. If a randomized scheduler is selected,
then the user can specify an integer weight for each transition in the determinator.
Figure 4-13 gives an example of a determinator that specifies weights for transitions

```
simulate Adder
transitions
if ready then
   result(value)
else
   add(1, 2) weight 1,
   add(2, 3) weight 2
```

Figure 4-13: Specifying weights for a randomized scheduler

when more than one transition is selected. This determinator specifies that if ready is
false, then transition add(2, 3) should be selected twice as frequently as transition
add(1, 2). If a time-based scheduler is selected, then the user must specify the time
estimates for transitions when more than one transition is selected.

An example of a determinator with time estimates is given in Figure 4-14. This
determinator specifies that transition add(2, 3) takes approximately twice the time
of transition add(1, 2); thus, transition add(1, 2) should be selected twice as fre-
quently as transition add(2, 3). The reader is referred to Section 5.4.1 for a detailed
discussion of how these schedulers work.

The simulator produces a log that contains a description of the initial state of the
simulated automaton and a trace of the executed actions. The user is able to specify
the name of a log file by using the standard UNIX input/output redirection. If a
simulate Adder
  transitions
    if ready then
      result(value)
    else
      add(1, 2) time 1,
      add(2, 3) time 2

Figure 4-14: Specifying time estimates

name for a log file is not specified, then the simulator writes the log to the standard output.

The user can also specify the total number of actions performed during the simulation using the `step` flag followed by the number of simulation steps. The number of actions performed between two consecutive dumps of state information can be specified using `acts` flag followed by the number.

Figure 4-15 gives an example of an execution trace of automaton Adder from Figure 3-1 with the determinator in Figure 4-5. The simulation was run for five steps and the state information was recorded after every transition. The randomized scheduler with equal weights was used. The simulator was invoked using the following command line:

```java
java larch.simulator -rand -step 5 -acts 1 < Adder.il > Log.sim.
```

4.5 Example of Simulating a Leader Election Algorithm

In this section we simulate a distributed algorithm that solves the leader election problem in networks with a ring topology (the algorithm was described in Section 3.3). We give a modified IOA description of the algorithm as an explicit composition of I/O process and channel automata, we state an invariant for the algorithm, and present a determinator and sample execution runs for the algorithm.
Simulation of automaton Adder
states:
  ready = false
  value = 0
actions:  add(1, 2)
states:
  ready = true
  value = 3
actions:  add(3, 4)
states:
  ready = true
  value = 7
actions:  result(7)
states:
  ready = false
  value = 7
actions:  add(7, 8)
states:
  ready = true
  value = 15
actions:  add(15, 16)
states:
  ready = true
  value = 31
simulation stopped: executed 5 transitions

Figure 4-15: A sample execution run of automaton Adder
There are several difficulties in simulating the automaton LCR as given on Figure 3-6. The first difficulty is in determining and scheduling enabled transitions at each step of the simulation. The determinator is used to solve this problem. Second, we need to provide an implementation for the non-built-in data type RingIndex used in the description of automaton LCR, giving a particular implementation for the operators uid, first, right, and left. Instead of implementing the RingIndex data type, we modify the specification for LCR to use Int for the Index data type, and we parameterize the Process automaton with n, the number of processes in the ring. This enables us to count modulo n in order to impose a ring structure on process indices. We also provide a uid to the Process automaton as a parameter, rather than using a fixed function from indices to uids. This allows us to experiment with different distributions of process uids in the ring and obtain different communication patterns during the simulation of the LCR algorithm. Third, we need to instantiate the Process and Channel automata with particular indices. As described in Section 4.2, the current implementation of the simulator requires that each automaton be instantiated separately. Therefore, in the description of the automaton LCR3, we instantiate each Process and Channel without using parameterized composition.

The modified IOA description of the process automata is presented in Figure 4-16. The automaton Process has three integer parameters. Parameter i is the index of the process in the ring graph G, and parameter n is the number of processes in the ring G. The parameter uid indicates the unique identifier of process i. We use mod(i-1, n) instead of left(i) and mod(i+1, n) instead of right(i). Channel automata are the same as in Figure 3-5, except that they do not have the parameter indicating the type of process indices, and the indices i and j are integers.

Figure 4-18 presents an automaton, LCR3, which is describes the algorithm in a ring of three processes as a composition of process and channel I/O automata. Figure 4-19 restates the invariant shown in Figure 3-7 for automaton LCR3, in order to eliminate the use of the universal quantifiers.

Figure 4-20 gives an example of a determinator for automaton LCR3. Since LCR3 does not have choose variables, there is no choose part in the determinator. The
automaton Process(i, uid, n: Int)

type Status = enumeration of waiting, elected, announced

signature
  input receive(m: Int, const mod(i - 1, n), const i)
  output send(m: Int, const i, const mod(i+1, n)),
         leader(m: Int, const i)

states
  pending: Mset[Int] := {uid},
  status: Status := waiting

transitions
  input receive(m, j, i)
  eff if m > uid then pending := insert(m, pending)
         elseif m = uid then status := elected
  fi
  output send(m, i, j)
  pre m ∈ pending
  eff pending := delete(m, pending)

output leader(m, i)
  pre status = elected ∧ m = uid
  eff status := announced

tasks
  {send(m, j, mod(j+1, n)) for m, j: Int};
  {leader(m, j) for m, j: Int}

Figure 4-16: Modified IOA specification of election process

automaton Channel(i, j: Int)

signature
  input send(m: Int, const i, const j)
  output receive(m: Int, const i, const j)

states
  buffer: Seq[Int] := {}

transitions
  input send(m, i, j)
  eff buffer := buffer ⊕ m

output receive(m, i, j)
  pre buffer ≠ {} ∧ m = head(buffer)
  eff buffer := tail(buffer)

Figure 4-17: Modified IOA specification of channel automaton
automaton LCR3
compose
Process(0, 11, 3); Process(1, 8, 3); Process(2, 15, 3);
Channel(0, 1); Channel(1, 2); Channel(2, 0)

Figure 4-18: IOA specification for LCR algorithm using three processes

invariant of LCR3:
\[
\begin{align*}
\text{Process}(0, 11, 3).status &= \text{elected} \Rightarrow \\
&\quad ( \text{Process}(1, 8, 3).status \neq \text{elected} \\
&\quad \quad \land \text{Process}(2, 15, 3).status \neq \text{elected} ) \\
\land \text{Process}(1, 8, 3).status &= \text{elected} \Rightarrow \\
&\quad ( \text{Process}(0, 11, 3).status \neq \text{elected} \\
&\quad \quad \land \text{Process}(2, 15, 3).status \neq \text{elected} ) \\
\land \text{Process}(2, 15, 3).status &= \text{elected} \Rightarrow \\
&\quad ( \text{Process}(1, 8, 3).status \neq \text{elected} \\
&\quad \quad \land \text{Process}(0, 11, 3).status \neq \text{elected} )
\end{align*}
\]

Figure 4-19: IOA specification of invariant for states of automaton LCR3

simulate LCR3
transitions
if Channel(0, 1).buffer = {} then
    send(i, 0, 1) for i: Int in Process(0, 11, 3).pending
else receive(head(Channel(0, 1).buffer), 0, 1)
if Channel(1, 2).buffer = {} then
    send(i, 1, 2) for i: Int in Process(1, 8, 3).pending
else receive(head(Channel(1, 2).buffer), 1, 2)
if Channel(2, 0).buffer = {} then
    send(i, 2, 0) for i: Int in Process(2, 15, 3).pending
else receive(head(Channel(2, 0).buffer), 2, 0)
if Process(0, 11, 3).status = elected then leader(11, 0)
if Process(1, 8, 3).status = elected then leader(8, 1)
if Process(2, 15, 3).status = elected then leader(15, 2)

Figure 4-20: Determinator for automaton LCR3
transitions part specifies the scheduling of transitions during simulation of LCR3. In this case, the determinator specifies every enabled transition definition and its parameters at every state.

Figure 4-21 presents a sample execution run of automaton LCR3. In this run the simulator was asked to report state information after executing every five actions. Since the specification for the automaton LCR3 contains an invariant, the invariant was checked at every step. An equal weight randomized scheduling algorithm was used for scheduling action executions. The simulation stopped when no actions were enabled, that is, after the maxim uid traveled around the ring and the process with the maximum uid executed the leader(15, 2) action.

Figure 4-22 presents a determinator that uses a timed scheduling algorithm, to be described in Section 5.4.1. This determinator is the same as the one in Figure 4-20, but contains timing estimates for the send and receive actions, making Channel(2, 0) slow and Channel(1, 2) fast. Figure 4-23 gives a sample execution of automaton LCR3 using the time-based scheduling. In this execution several pileups occurred in the channel, since some uids traveled faster then others.

4.6 Performance

Table 4.1 presents information about the simulation of the LCR algorithm with 3, 5, 10, and 20 processes and channels. The table gives the number of lines of the source files for the intermediate representation and the simulation time for each automaton. The source files for the intermediate representation are large because they include the symbol table for all operations of all data types used by the automata (see Section 5.3). The simulation was done on a PC with a 200 MHz Pentium Pro processor running the Red Hat 4.2 version of UNIX OS. The LCR algorithm with 20 processes was simulated in less than half a second.
Simulation of automaton LCR3

states:
  Process(0, 11, 3).pending = Mset: {11}
  Process(0, 11, 3).status = waiting
  Process(1, 8, 3).pending = Mset: {8}
  Process(1, 8, 3).status = waiting
  Process(2, 15, 3).pending = Mset: {15}
  Process(2, 15, 3).status = waiting
  Channel(0, 1).buffer = Seq: {}
  Channel(1, 2).buffer = Seq: {}
  Channel(2, 0).buffer = Seq: {}

actions:
  send(8, 1, 2), receive(8, 1, 2), send(11, 0, 1), receive(11, 0, 1),
  send(11, 1, 2)

states:
  Process(0, 11, 3).pending = Mset: {}
  Process(0, 11, 3).status = waiting
  Process(1, 8, 3).pending = Mset: {}
  Process(1, 8, 3).status = waiting
  Process(2, 15, 3).pending = Mset: {15}
  Process(2, 15, 3).status = waiting
  Channel(0, 1).buffer = Seq: {}
  Channel(1, 2).buffer = Seq: {11}
  Channel(2, 0).buffer = Seq: {}

actions:
  receive(11, 0, 1), send(11, 1, 2), send(15, 2, 0), receive(11, 1, 2),
  receive(15, 2, 0)

states:
  Process(0, 11, 3).pending = Mset: {}
  Process(0, 11, 3).status = waiting
  Process(1, 8, 3).pending = Mset: {15}
  Process(1, 8, 3).status = waiting
  Process(2, 15, 3).pending = Mset: {}
  Process(2, 15, 3).status = waiting
  Channel(0, 1).buffer = Seq: {}
  Channel(1, 2).buffer = Seq: {}
  Channel(2, 0).buffer = Seq: {}

actions:
  send(15, 1, 2), receive(15, 1, 2), leader(15, 2)

simulation stopped: no enabled actions

Figure 4-21: A sample execution run of automaton LCR3
simulate LCR3

transitions
if Channel(0, 1).buffer = {} then
    send(i, 0, 1) for i: Int in Process(0, 11, 3).pending time 3
else receive(head(Channel(0, 1).buffer), 0, 1) time 3
if Channel(1, 2).buffer = {} then
    send(i, 1, 2) for i: Int in Process(1, 8, 3).pending time 1
else receive(head(Channel(1, 2).buffer), 1, 2) time = 1
if Channel(2, 0).buffer = {} then
    send(i, 2, 0) for i: Int in Process(2, 15, 3).pending time 10
else receive(head(Channel(2, 0).buffer), 2, 0) time = 10
if Process(0, 11, 3).status = elected then leader(11, 0) time 3
if Process(1, 8, 3).status = elected then leader(8, 1) time 1
if Process(2, 15, 3).status = elected then leader(15, 2) time 10

Figure 4-22: Determinator with time estimates for automaton LCR3

4.7 Related Work

The IOA language has evolved from earlier work in Professor Lynch's research group on describing distributed algorithms in the form of pseudocode. Goldman in his Ph.D. thesis on the Spectrum system defined a programming language with preconditions and effects, but without any nondeterministic constructs. He designed a simple simulator for that language. The design of the Spectrum language was specifically tuned for the simulator: for example, each transition definition had a SEL field, where the user had to specify transition parameters for the simulation. In contrast, the IOA system isolates the process of automaton definition from the process of simulating an automaton. In addition, the Spectrum language did not have constructs for randomization and user definition of parameters (although the Spectrum user interface allowed some of this functionality), while the IOA simulator has the RAND and USER functions in its determinator language.

Goldman's more recent work on the Programmers' Playground [11] includes software support and formal semantics in terms of the I/O automaton model for designing distributed applications; however, it has no facilities for simulation and proofs. Instead, the Programmer's Playground provides support for debugging distributed
Simulation of automaton LCR3

states:
Process(0, 11, 3).pending = Mset: {11}
Process(0, 11, 3).status = waiting
Process(1, 8, 3).pending = Mset: {8}
Process(1, 8, 3).status = waiting
Process(2, 15, 3).pending = Mset: {15}
Process(2, 15, 3).status = waiting
Channel(0, 1).buffer = Seq:{}
Channel(1, 2).buffer = Seq:{}
Channel(2, 0).buffer = Seq:{}

actions:
send(11, 0, 1), send(15, 2, 0), send(8, 1, 2), receive(15, 2, 0),
send(15, 0, 1)

states:
Process(0, 11, 3).pending = Mset: {}
Process(0, 11, 3).status = waiting
Process(1, 8, 3).pending = Mset: {}
Process(1, 8, 3).status = waiting
Process(2, 15, 3).pending = Mset: {}
Process(2, 15, 3).status = waiting
Channel(0, 1).buffer = Seq:{11, 15}
Channel(1, 2).buffer = Seq:{8}
Channel(2, 0).buffer = Seq:{}

actions:
receive(11, 0, 1), send(11, 1, 2), receive(15, 0, 1), send(15, 1, 2),
receive(8, 1, 2)

states:
Process(0, 11, 3).pending = Mset: {}
Process(0, 11, 3).status = waiting
Process(1, 8, 3).pending = Mset: {}
Process(1, 8, 3).status = waiting
Process(2, 15, 3).pending = Mset: {}
Process(2, 15, 3).status = waiting
Channel(0, 1).buffer = Seq:{}
Channel(1, 2).buffer = Seq:{11, 15}
Channel(2, 0).buffer = Seq:{}

actions:
receive(11, 1, 2), receive(15, 1, 2), leader(15, 2)

simulation stopped: no enabled actions

Figure 4-23: A sample execution run of automaton LCR3 using time-based scheduling
Table 4.1: Simulator's performance for LCR algorithm

<table>
<thead>
<tr>
<th>No. of processes</th>
<th>3</th>
<th>5</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of lines in intermediate spec</td>
<td>298</td>
<td>316</td>
<td>348</td>
<td>392</td>
</tr>
<tr>
<td>Simulation Time(msec)</td>
<td>120</td>
<td>150</td>
<td>190</td>
<td>230</td>
</tr>
</tbody>
</table>

applications by monitoring so called “published” variables. Cheiner and Shvartsman [3] gave some suggestions on general strategies for resolving nondeterminism in I/O automata using levels of abstractions; their work, however, concerned code generation and not simulation.
Chapter 5

Implementation

Since the simulator is a part of the IOA system, the design and implementation of the whole IOA system and the simulator for the IOA language are interdependent. We start this chapter by describing the IOA toolset architecture. We then discuss the IOA front-end tools for transforming IOA programs into an intermediate form, which is used by the IOA back-end tools. We present the design and implementation of the intermediate language. Finally, we describe the implementation of the simulator and the libraries of scheduling policies and basic data types supported by the simulator.

5.1 IOA Tools

As described in Section 1.2, the IOA system provides a variety of tools for the production of high-quality distributed software. The user is provided with a combination of analytical tools, which are designed to complement each other in verification and debugging. IOA specifications can also be translated automatically into corresponding Java or C++ code.

An interface to a theorem prover can be used to prove invariants, simulation relations, and the validity of systems described using IOA.

An interface to a model checker can be used to exhaustively test an automaton’s properties in all reachable states, provided the number of states is finite and sufficiently small.
The simulator allows the user to observe the run-time behavior of an IOA program; it can be used for debugging and testing the correctness and performance of a distributed algorithm. The design and implementation of the simulation tool are described in detail in Section 5.4.

A code generator generates real code for a distributed system. The code generator and the simulator need to resolve many similar issues when converting axiomatic description into running code. In order to resolve nondeterminism of IOA programs, the code generator can use the determinator described in Section 4.3. It can also reuse the library of basic data types described in Section 5.4.2.

\section*{5.2 IOA Front End}

The IOA front end tools consist of a parser, syntactic and semantic checker, prettyprinters, and transformation tools.

The IOA parser parses, reports syntax errors, and constructs an internal representation for IOA specifications. The parser is written using the Java CUP \cite{javacup} LALR parser generator and is implemented using standard compiler techniques (see \cite{编译原理}). The grammar together with the static semantic checks for IOA are described in \cite{IOAgrammar}. One prettyprinter indents the code consistently and breaks long lines; another translates IOA specifications into \LaTeX{} format.

In order to be used as input to some of the IOA tools, IOA programs are first converted into a simplified intermediate representation. The IOA front end provides a tool to transform IOA programs into the intermediate language, which serves as input language to the IOA toolset and drives the second tier of the IOA system architecture (see Figure 1-1).

Two transformations that occur in the translation into the intermediate representation are composition and term expansion. Composition expansion transforms the definition of a composite automaton into a single I/O automaton definition without the \texttt{compose} operator. This transformation is performed in the front-end, and composite automata do not appear in the intermediate language, to avoid duplicate
effort in the back-end tools. Term expansion is applied to actions in the automaton’s signature; it replaces action parameters written as expressions by fresh variables constrained by \texttt{where} clauses. This transformation is needed because, in the intermediate language, action parameters in the automaton’s signature are required to be variables in order to combine the \texttt{where} constraints for the same action header in different component automata. In generating the intermediate representation for an IOA description of an automaton, the term expansion transformation is applied before composition expansion. We describe term and composition expansions in Sections 5.2.1 and 5.2.2.

5.2.1 Term Expansion

In the IOA specification of an automaton’s signature, an action parameter can be specified as either a variable or a term. However, in the intermediate language all action parameters in the signature are required to be variables.

Thus, during translation into the intermediate language, whenever an action parameter is represented as a term, \textit{term expansion} is applied. This transformation is done on the parameters of an action in the automaton’s \textit{signature}. A corresponding transformation on the parameters of the automaton’s transitions is more complicated. It is discussed in Section 7.2 and is not currently implemented. Term expansion transforms action parameters specified as terms into fresh variables and corresponding restrictions on these variables.

When specifying the signature of an automaton in IOA, the \texttt{const} operator can be used to indicate that the value of an action parameter is fixed by the value of a given term (see Section 2 of the IOA manual [9]). The operator \texttt{const} provides syntactic sugar for a \texttt{where} clause. The transformation is done by rewriting \texttt{const} clauses in the IOA source code as the corresponding \texttt{where} clauses using the following desugaring function \( D \):

\[
D[(\text{actionName})(\text{const } t)] = (\text{actionName})(f : T) \text{ where } f = t.
\]
Here `<actionName>` stands for the name of an action in the automaton’s signature, T is the type of the term t and f is a fresh variable of type T. For example, `blah(const 2)` is expanded to `blah(i:Int) where i = 2`.

### 5.2.2 Composition Expansion

The IOA language allows a finite collection of I/O automata to be composed into one automaton using a `compose` operation. We decided that the IOA intermediate language should not include this composition operation. If we had included a composition operator in the intermediate language instead of implementing the composition transformation, each IOA tool would have to perform a variant of the composition transformation. We chose to do the composition transformation at the IOA language level to ensure that all tools use the same notion of composition.

The front-end tool that implements the composition expansion, the `composer`, is described in detail in Chapter 6.

### 5.3 Intermediate Language

The intermediate language is the input language to the IOA toolset (see Figure 1-1). It is a simplified variant of the IOA language.

Appendix A includes the BNF grammar for the intermediate language. The intermediate representation of an IOA specification contains two parts: a global symbol table and a list of intermediate forms for primitive automata specification (see Section 3.4). The global symbol table consists of types and operators that are used by all automata in the specifications. It does not contain variables, since IOA has no global variables.

An automaton’s invariants are a part of the intermediate specification for the automaton. Each automaton is defined by its symbol table and an S-expression for the automaton definition, with resolved references for all variables, types, and operators. The symbol table defines all variables, types, and operators that are used in the automaton’s definition, but not declared in the global symbol table. In the
global and automaton's symbol tables, each variable, type, or operator is assigned a unique internal name for ease of reference. The symbol table also contains an external name, that is, the name used in the original IOA specification, for each variable, type, or operator. This design is simple and efficient: it significantly reduces the type checking of an automaton definition, while allowing the user of an IOA tool to refer to automaton variables, types, and operators by their original names.

Figure 5-1 gives an example of the intermediate representation of the automaton Adder defined in Figure 3-1. A variable declaration starts with the keyword var followed by a unique identifier for its internal name, and the external name and the type of the variable. A type declaration is indicated by the keyword sort followed by the internal and external names of the type. The keyword LITERAL in a type declaration indicates that the type is a literal type (that is, decimal literals are the constructors of the type). An operator declaration is indicated by the keyword op followed by the operator's internal name, its external name, and its signature. The internal names for variables, types, and operators are used in the description of the automaton's actions, states, transitions, and tasks.

If the IOA specification for an automaton uses the compose operator, then its intermediate representation is that of the primitive automaton description obtained from applying the composition expansion (see Section 5.2.2) to the IOA specification. In this representation, variables are described using references to state variables in the component automata. A reference to an automaton's state variable consists of the following three components:

1. A sequence of names for the component automata in which this state variable is defined, called a sequence of defining automata. If $S = S_1, S_2, ..., S_n$ is a sequence of defining automata for a state variable $x$, then automaton $S_n$ should have a primitive automaton description, and for all $i \in \{2, ..., n\}$ automaton $S_i$ should be a component of the composite automaton $S_{i-1}$.

2. A sequence of actual parameters for every parameterized automaton in the defining sequence.
/*top-level scope*/
(sort s1 "Bool")
(op o11 "\(_\lor\(_\" s1 s1 s1)
(op o7 "false" s1)
(op o12 "true" s1)
...
(op o2 "if" s1 s0 s0 s0)
(automaton Adder

/*scope for automaton Adder*/
(sort s3 "Int" LITERAL)
(var v4 "ready" s1)
(var v5 "i" s3)
(var v2 "k" s3)
(var v0 "i" s3)
(var v7 "k" s3)
(var v3 "value" s3)
(var v6 "j" s3)
(var v1 "j" s3)
(op o33 "max" s3 s3 s3)
(op o31 "mod" s3 s3 s3)
...
(op o37 "\(_\geq\(_\" s3 s3 s1)

/*description of automaton Adder*/
(action input add v0 v1)
(action output result v2)
(state v3)
(state v4 o7)
(transition input add (actuals v5 v6)
  (eff ((assign v3 (o23 v5 v6)) (assign v4 o12))))
(transition output result (actuals v7) (pre (o6 (o294 v7 v3) v4))
  (eff ((assign v4 o7))))
)

Figure 5-1: Intermediate representation of automaton Adder
3. A name of a state variable.

For example, the reference \( C(1).B(3, \text{true}).A.x \) tells that \( x \) is a state variable in automaton \( A \), which is a component of automaton \( B \) instantiated with actual parameters \( 3 \) and \( \text{true} \), which in turn is a component of automaton \( C \) instantiated with actual parameter \( 1 \). In the intermediate language, a reference to state variable \( C.B(2).A(3, 5).x \) is represented as \((2, 3, 5) C.B(\text{Int}).A(\text{Int}, \text{Int}) \, x\).

Figure 5-2 gives an example of the intermediate representation of the composite automaton \( \text{LCR3} \) defined in Figure 4-18.

If there are several transition definitions for a particular action, the intermediate representation of the automaton numbers them in the order of their appearance.

The parser and static semantic checker for the intermediate language are simple and are implemented using JavaCC [16], a recursive descent parser generator. Since the symbol table for a specification is constructed before parsing the specification, type checking is easy.

### 5.4 Implementation of the Simulator

The input to the simulator consists of the intermediate representation for an automaton and a determinator. The simulator provides the user with scheduling options. By default, the simulator uses randomized scheduling with equal probabilities for all selected actions. At every step of the simulation, the set of possible transitions and their parameter values is given by the determinator. If more than one transition is possible, the scheduler picks one according to the scheduling policy selected by the user before the simulation began. If both the precondition clause and the \textit{where} constraint of the chosen transition definition and its parameter values evaluate to true in the latest simulated state, then the effect is executed; otherwise, a failure message is logged into the execution trace and the user is informed about the failure. After execution of the chosen transition, the state invariants are checked, and the execution trace is updated. Figure 5-3 presents pseudocode for the main loop of the simulator.
/*top-level scope*/
(sort s1 "Bool")
(op o2 "if" s1 s0 s0 s0)
(op o7 "false" s1)
...
(automaton LCR3
/*scope for automaton LCR3*/
(sort s24 "Status")
(sort s3 "Int" LITERAL)
(sort s27 "Mset" s3)
(sort s23 "Seq" s3)
...
(var v25 "Process(1, 8, 3).pending" s27)
(var v37 "Process(2, 15, 3).status" s24)
...
(op o402 "last" s23 s3)
(op 30 "--*--" s3 s3 s3)
/*description of automaton LCR3*/
(action output receive v3 v26 v31 (where
  (o11
   (o11 (o6 (o406 v39 0:s3) (o406 v40 1:s3))
      (o6 (o406 v26 1:s3) (o406 v31 2:s3))
      (o6 (o406 v26 2:s3) (o406 v31 0:s3))))
...
(state v48 (o345 11:s3))
(state v37 o318)
...
(transition input receive (actuals v10 v23 v25)
  (eff
   ((if ((o6 (o406 v23 (o31 (o28 2:s3 1:s3) 3:s3) 3:s3))
     (o406 v25 2:s3))
   ((if ((o36 v10 15:s3)
     ((assign (o18 2:s3 15:s3 3:s3))
       (o341 v10 (o103 2:s3 15:s3 3:s3))
...

Figure 5-2: Intermediate representation of automaton LCR3
while true {
    determine the set of enabled actions using the determinator
    if this set is not empty then
        get next transition definition from scheduler
        if both precondition and where clause of
            selected transition evaluate to true
            then assign values to choose parameters of selected transition
            execute effect clause of selected transition
            check invariants
            report and log executed transition and current state
        else report and log failure message and exit
    else stop simulation and exit
} 

Figure 5-3: Pseudocode for simulator's main loop

5.4.1 Schedulers

The job of the scheduler is to select the next transition to be executed based on the current state and the determinator, which specifies the set of actions that can be performed. The scheduler does not determine all enabled actions; only the actions specified by the determinator are considered for execution. If the determinator specifies more than one action in a state, then the scheduler selects one according to a scheduling policy specified by the user before the simulation began. The user has a choice of three scheduling policies: randomized, round robin, and one based on time estimates for each action. Since the design of the simulator is modular, it should not be difficult to add other scheduling policies.

The randomized scheduler makes use of weights on transitions (see Section 4.4). It computes the total $t$ of the weights of all specified transitions, and at each step of the execution selects a transition having weight $w$ with probability $w/t$. The round robin scheduler keeps count of the number of times a transition was specified but not selected for execution and maintains a queue of these counts. It always selects the transition with the greatest count. The count is reset to zero after the transition is executed or is not selected.

The system provides assistance in coming up with probabilities for each action,
by allowing the user to provide a *time estimate* for each action (see Section 4.4). The time estimate for an action can be used to schedule the action according to the speed of the processor on which it is intended to run. The time estimates can also be used to account for computation latency or the rate at which an environment generates actions. The smaller the time estimate, the faster the processor, and the higher the probability that the action will get scheduled. Thus, actions running on a fast processor will get executed more frequently. The scheduler determines the probabilities based on the user-specified time estimates, with actions with low time estimates having correspondingly high probabilities. Specifically, if times for \( n \) actions are given by \( n \) integers \( \text{time}_0, \text{time}_1, \ldots, \text{time}_{n-1} \), then the scheduler determines which of the \( n \) actions to perform by the following procedure:

1. Find the least common multiple (lcm), \( m \), for the set \( \{ \text{time}_i : 0 \leq i \leq n - 1 \} \).

2. Assign a weight to each selected action as follows:

\[
weight_i = \frac{(m/\text{time}_i)}{\sum_{j=0}^{n-1}(m/\text{time}_j)}.
\]

3. Apply the general weighted-random scheduler to the collection of actions with weights \( \{ weight_i : 0 \leq i \leq n - 1 \} \). That is, divide the interval [0..1] into \( n \) parts

\[
[0..weight_0], [weight_0..weight_0 + weight_1], \ldots, [\sum_{j=0}^{n-2} weight_j .. 1]
\]

and schedule the next action be performed to be action \( i \) if the random number is in the range

\[
[\sum_{j=0}^{i-1} weight_j \ldots \sum_{j=0}^{i} weight_j].
\]

This scheduler assigns weights \( m/\text{time}_i \) that are inversely proportional to the time estimates for the actions. The lcm, \( m \), is used to avoid floating point computation and to reduce the round-off errors.

### 5.4.2 Data Types

Data types are defined axiomatically in IOA, but in order to simulate data type operations, the simulator needs actual code for those operations. The simulator has a
library of data type definitions written in Java. Several IOA built-in data types translate straightforwardly into corresponding Java data types and operations on them: booleans (Bool), integers (Int), natural numbers (Nat), real numbers (Real), characters (Char), and strings (String). The IOA built-in type constructors for one dimensional arrays (Array[type]), sets (Set[type]), multisets (Mset[type]), sequences (Seq[type]), and mappings (Map[type]) are implemented using Java Objects. The simulator also supports data types that are defined using LSL enumeration, tuple, or union.

We have written a library for all built-in operators. This code is written in accordance with the LSL specification for the corresponding trait defined in [14]. If the IOA specification of an automaton description uses a non-built-in data type, then in order to be able to simulate this automaton the user must provide a Java implementation for this data type. The implementation of all operators for a non-built-in data type should be supplied in a class named by the name of the data type. For example, if the user wants to use a positive integer type, Pos, in an IOA specification, then he/she writes an LSL trait for the Pos type. If the user wants to simulate this automaton, he/she must provide implementations for all the operators for type Pos. The simulator dynamically loads the implementations of all data types used in a given automaton. If an implementation for a data type is not found, then the simulator informs the user of the failure and stops the simulation.

The simulator stops the simulation if one of the following run-time errors is encountered:

1. Division by zero.

2. The succ function is applied to the last element in a sequence. For example, the following IOA specification will generate this run-time error during simulation:
type Color = enumeration of white, red, black

automaton A
signature output foo
states c: Color := black
transitions
  output foo
  eff c := succ(c)

3. Attempt to iterate over a set with a so that constraint, since, in general, the simulator cannot determine the set and/or the set can be infinite. For example, the following IOA specification will generate a run-time error during simulation:

automaton A
signature output foo
states x: Array[Int]
transitions
  output foo
  eff for i: Int so that i > 0 do
    x[i] := i
  od

5.5 Design decisions

We decided to both write the simulator in Java and generate Java source code for IOA, because Java is portable and relatively easy to code and debug. We considered two alternatives to Java: C and C++. The main advantage of C and C++ would be the speed of the simulator and the generated code; efficiency is an important design goal for a simulator. However, in view of the current development of efficient Java compilers and Java processors\(^1\), we decided that the difference in code efficiency would not be significant. The simulator and the parser for the intermediate language are

\(^1\)For example, Cygnus is writing an optimized Java front end for gcc, and Sun has designed a highly efficient Java processor, JavaChip.
implemented in less than 6,000 lines of Java code. The code for the simulator can be found in http://www.sds.lcs.mit.edu/~achefter/Simulator/. Profiling tests of the simulator show that most of the time is spent in the Java system routines for memory garbage collection. Since JavaSoft is now implementing more efficient garbage collectors, the simulator will also become faster.
Chapter 6

Composer

In this chapter, we describe the composer, an IOA front end tool that, given an IOA description of a composite automaton, constructs an equivalent primitive IOA automaton description for the automaton. The semantics of this composition transformation conform to that of the composition operation described in Section 2.2. The composer is used for translating IOA descriptions of composite automata into the intermediate language. If any of the components are composite automata, then the composition transformation is applied recursively to these components.

6.1 Input to the Composer

We describe the composition transformation by showing its effects on a general automaton specification A, shown in Figure 6-1. Its component automata Ai are assumed to have primitive IOA automaton descriptions. If a component automaton Ai has formal parameters, then the formals of Ai are replaced by the actuals provided in the definition of the composite automaton A.

For the automata Ai to be compatible the where clauses should satisfy the property that for all values of the parameters i1, ..., ik at most one of the Qi is true. To guarantee that no transition is defined twice, it must be true that
automaton $A_i(p_1: T_1, \ldots, p_{v_i}: T_{v_i})$

signature

input name($i_1: I_1, \ldots, i_k: I_k$) where $P_i(i_1, \ldots, i_k), \ldots$

output name($i_1: I_1, \ldots, i_k: I_k$) where $Q_i(i_1, \ldots, i_k), \ldots$

states

$s_1: S_1,$

\ldots

$sp: S_p$

so that $S_i(s_1, \ldots, sp)$

transitions

input name($i_1, \ldots, i_k$) where $P_i,1(i_1, \ldots, i_k)$

\quad eff \ldots

\ldots

input name($i_1, \ldots, i_k$) where $P_i,1_i(i_1, \ldots, i_k)$

\quad eff \ldots

output name($i_1, \ldots, i_k$) where $Q_i,1(i_1, \ldots, i_k)$

\quad pre $R_i,1$

\quad eff \ldots

\ldots

output name($i_1, \ldots, i_k$) where $Q_i,m_i(i_1, \ldots, i_k)$

\quad pre $R_i,m_i$

\quad eff \ldots

\ldots

tasks

$\{\text{name}(i_1, \ldots, \text{in}) \text{ for } i_1: I_1, \ldots, i_k: I_k\}$

\ldots

automaton $A$

compose $A_1(a_1, 1, \ldots, a_1,v_1)$;

$A_2(a_2,1, \ldots, a_2,v_2)$;

\ldots;

$A_n(a_n, 1, \ldots, a_n,v_n)$

Figure 6-1: General input to composer
for all $i \in \{1, \ldots, n\}$
for all $j_1, j_2 \in \{1, \ldots, l_i\}$

$P_{i,j_1}(i_1, \ldots, i_k) \land P_{i,j_2}(i_1, \ldots, i_k) \Rightarrow j_1 = j_2$

and

for all $i \in \{1, \ldots, n\}$
for all $j_1, j_2 \in \{1, \ldots, m_i\}$

$Q_{i,j_1}(i_1, \ldots, i_k) \land Q_{i,j_2}(i_1, \ldots, i_k) \Rightarrow j_1 = j_2$.

To guarantee that all transitions are defined, we need to know that

for all $i \in \{1, \ldots, n\}$
for all $i_1, \ldots, i_k$

$P_i(i_1, \ldots, i_k) \Rightarrow \exists j \in \{1, \ldots, l_i\} \ P_{i,j}(i_1, \ldots, i_k)$

and

for all $i \in \{1, \ldots, n\}$
for all $i_1, \ldots, i_k$

$Q_i(i_1, \ldots, i_k) \Rightarrow \exists j \in \{1, \ldots, m_i\} \ Q_{i,j}(i_1, \ldots, i_k)$.

In the following sections we describe a specification for a primitive automaton description $A_{\text{Expanded}}$ that corresponds to the IOA specification of the composite automaton $A$. The state variables of the automaton $A_{\text{Expanded}}$, as described in Section 6.3, cannot be parsed by the IOA parser and are used only for describing the composition rule. This is not a problem because these state variables have a representation in the IOA intermediate language, as described in Section 5.3.

### 6.2 Signature of Automaton $A_{\text{Expanded}}$

If a component automaton $A_i$ has formal parameters, then in the signature of $A_{\text{Expanded}}$ the formals of $A_i$ are replaced by the actuals provided in the compose clause in the definition of the composite automaton $A$. For example, in Figure 6-2 the automaton $A_1$ has a formal type parameter $T$. In the definition of the composition
automaton A, A1 has been instantiated as A1(String). Therefore, in the signature of AExpanded, all occurrences of T in A1’s signature are replaced by String, resulting in the action header input foo(m: String).

automaton A1(T: type)
  signature
  input foo(m: T)
  states
  value: T
  transitions
  input foo(m)
  eff value := m
automaton A
  compose A1(type String); ...

Figure 6-2: Example of instantiation of type parameters

6.2.1 Output and Internal Actions of Automaton AExpanded

An action header name(i1:I1, ..., ik:Ik) where P(i1, ..., ik) is included in the output (or internal) action list in the signature of AExpanded provided that the pattern name(I1, ..., Ik) appears in at least one of Ai’s output (internal) action lists and

\[ P = (P_1 \lor P_2 \lor ...). \]

Here, the Pi are the where clauses of the action headers with pattern name(I1, ..., Ik) that appear in the output (internal) action lists in the component automata.

If one of the action headers of Ai’s output (internal) action lists does not have a where clause, i.e., if one of the Pi is always true, then the where clause for the corresponding action header in the signature of AExpanded is omitted (since P simplifies to true). See Figure 6-3 for examples of the signature of automaton AExpanded.
6.2.2 Input Actions of Automaton AExpanded

An action header \texttt{name}(i1:i1, \ldots, ik:Ik) \texttt{where} P(i1, \ldots, ik) is included in the input action list in the signature of \texttt{AExpanded} provided that an action header with action pattern \texttt{name}(I1, \ldots, Ik) appears in at least one of the Ai's input action lists and
\[
P = (P_1 \lor P_2 \lor \ldots) \land \neg(Q_1 \lor Q_2 \lor \ldots).
\]

Here, the \texttt{Pi} are the where clauses of the action headers with the pattern \texttt{name}(I1, \ldots, Ik) that appear in input action lists in the component automata, and the \texttt{Qi} are the where clauses of the action headers with the pattern \texttt{name}(I1, \ldots, Ik) that appear in output action lists in the component automata.

If one of the matched output action headers does not have a where clause, i.e., some Qi is always true, then the signature of the composite automaton does not include the matched action as an input action (since the where clause simplifies to \texttt{false}). Figure 6-3 gives an example of the described rule.

\begin{verbatim}
automaton A1
  signature
  input  foo(i: Int) where 1 \leq i \leq 10
  output blah(j: Int)

automaton A2
  signature
  input blah(i: Int) where P(i)
  output foo(j: Int) where j < 3

automaton A
  compose A1; A2

automaton AExpanded
  signature
  input foo(i: Int) where 1 \leq i \leq 10 \land \neg(i < 3)
  output foo(i: Int) where i < 3,
                 blah(j: Int)

Figure 6-3: Example of signature of automaton AExpanded
\end{verbatim}
6.3 States of Automaton AExpanded

The state variables of AExpanded contain those of each of Ai with the state reference (see Section 5.3) modified as follows:

1. The name of the automaton Ai is prepended to the sequence of defining automata.

2. The actual parameters of Ai are prepended to the sequence of actual parameters for the state reference in Ai.

The expressions describing the initial values of the state variables for Ai are modified by replacing the formal parameters of Ai with the corresponding actual parameters.

If some Ai has a so that restriction on its state’s initial values, then a so that clause for the state variables of the automaton AExpanded is constructed by expanding and forming the conjunction of the so that clauses for the state variables of Ai. Thus, the so that predicate S of automaton AExpanded can be expressed as

\[ S = S_1 \land S_2 \land \ldots \land S_n, \]

where the Si are the so that clauses for the state variables of the component automata, modified by expanding the state names used in their definition and by replacing Ai’s formal parameters with the corresponding actual parameters. For example,

automaton B(i: Int)
  signature ...
  states n: Int
    so that n > i
  transitions ...

automaton A
  compose B(1); B(2)

becomes
automaton AExpanded
signature ...
states B(1).n: Int,
   B(2).n: Int
   so that B(1).n > 1 ∧ B(2).n > 2
transitions ....

In systems that are described using several applications of the composition operation, this technique of prefixing variable names with automaton names can lead to long prefixes. IOA allows such prefixes to be abbreviated if there is no ambiguity.

6.4 Transitions of Automaton AExpanded

Automaton AExpanded has a single input transition definition and a single output transition definition for each action header name(i1, ..., ik) as shown in Figures 6-4 and 6-5. To avoid cluttering up the notation, we abbreviate (i1, ..., ik) as (j) in those figures.

input name(j)
  eff
  if P1,1(j) ∨ P1(j) then
    [effects of A1's transition definition 1 for input name(j)]
    fi;
  ...
  if P1,1_1(j) ∨ P1(j) then
    [effects of A1's transition definition 1_1 for input name(j)]
    fi;
  ...
  if Pn,1(j) ∨ Pn(j) then
    [effects of An's transition definition 1 for input name(j)]
    fi;
  ...
  if Pn,1_n(j) ∨ Pn(j) then
    [effects of An's transition definition 1_n for input name(j)]
    fi;

Figure 6-4: Input transition for composite automaton
output name(j)
pre
if Q1,1(j) ∧ Q1(j) then
    R1_1
elseif ...
elseif Q1,m_1(j) ∧ Q1(j) then
    R1,m_1
...
elseif Qn,1(j) ∧ Qn(j) then
    Rn_1
elseif ...
elseif Qn,m_n(j) ∧ Qn(j) then
    Rn,m_n

eff
if Q1,1(j) ∧ Q1(j) then
    [effects of A1’s transition definition 1 for output name(j)]
fi;
...
if Q1,m_1(j) ∧ Q1(j) then
    [effects of A1’s transition definition m_1 for output name(j)];
fi;
...
if Qn,1(j) ∧ Qn(j) then
    [effects of An’s transition definition 1 for output name(j)]
fi;
...
if Qn,m_n(j) ∧ Qn(j) then
    [effects of An’s transition definition m_n for output name(j)];
fi;
if P1,1(j) ∧ P1(j) then
    [effects of A1’s transition definition 1 for input name(j)]
fi;
...
if P1,l_1(j) ∧ P1(j) then
    [effects of A1’s transition definition l_1 for input name(j)]
fi;
...
if Pn,1(j) ∧ Pn(j) then
    [effects of An’s transition definition 1 for input name(j)]
fi;
...
if Pn,l_n(j) ∧ Pn(j) then
    [effects of An’s transition definition l_n for input name(j)]
fi;
...

Figure 6-5: Output transition for composite automaton

81
The transition for action header $\text{name}(i_1, \ldots, i_k)$ in the composition is formed by grouping together all transitions for $\text{name}(i_1, \ldots, i_k)$ of the component automata. For example, an input transition definition for the composition consists of a list of conditional statements. Each conditional statement corresponds to an input transition definition in a component automaton; its condition is the $\text{where}$ constraint and its consequence is the effects of that input transition. The input transition in the composition also inherits the $\text{where}$ clause

$$(P_1(j) \lor P_2(j) \lor \ldots \lor P_n(j)) \land \neg(Q_1(j) \lor Q_2(j) \lor \ldots \lor Q_n(j))$$

from the signature of automaton $A_{\text{Expanded}}$, as described in Section 6.2.2.

A similar procedure is applied to construct an output transition definition for the composition. Since output actions can have preconditions, the output transition definition of the composition incorporates the preconditions of the components.

The order of conditional statements in the transitions of automaton $A_{\text{Expanded}}$ does not matter, because the component automata are compatible and because there is at most one transition definition for each action in a component automaton. For the output transition definition, the compatibility assumption says that at most one condition is true, and thus only one effect of an output transition definition is executed.

Currently, the definition of IOA language requires that there be at most one transition definition for every action header and set of values for its parameters and action type (as explained at the beginning of this chapter). If this requirement is relaxed to allow a nondeterministic choice of transition definition, then we would need to rewrite the $\text{if...then}$ statements in the input and output transition definitions as $\text{if...then...else...}$ statements and introduce a $\text{choose}$ parameter for determining which one branch of the $\text{if...then...else}$ statements is picked for execution.

The IOA language requires that each action pattern occur only once in each of the input/output/internal action lists; therefore, in the signature of $A_{\text{Expanded}}$, we combine the $\text{where}$ constraints of the component automata to construct the constraint for an action of the composite automaton. The $\text{where}$ predicates of individual transition definitions are treated as constraints to be added to those already present.
the corresponding \textbf{where} predicate in the signature. We write

\begin{verbatim}
if P_{1,1}(j) \land P_1(j) then
  \text{[effects of A1's transition definition 1 for input name(j)]}
\end{verbatim}

in the Figures 6-4 and 6-5, instead of

\begin{verbatim}
if P_{1,1}(j) then
  \text{[effects of A1's transition definition 1 for input name(j)]}
\end{verbatim}

because $P_{1,1}(j) \land P_1(j)$ is the specified constraint on the action \texttt{name(i1, \ldots, ik)} in the component automaton $A_i$.

\section{Tasks of Automaton $A_{Expanded}$}

The set of tasks for $A_{Expanded}$ is the union of the sets of tasks of $A_1$, $A_2$, \ldots, $A_n$. If any of the component automata does not have a set of tasks explicitly specified, then its set of tasks is assumed to contain the single set of all its output and internal actions. Figure 6-6 gives an example. The first task of the automaton $A_{Expanded}$ is the set of locally controlled (that is, output and internal) actions of automaton $A_1$. The second and third tasks are those of automaton $A_2$. 

83
automaton A1
signature
  input  A1Input(i: Int) where 1 ≤ i ≤ 10
  output A1Output(j: Int)
  internal A1Internal(k: Int) where R(k)
...

automaton A2
signature
  input  A1Output(i: Int) where P(i)
  output A1Input(j: Int) where j < 3
  internal A2Internal(b: Bool)
...

tasks
  {A1Input(i) for i: Int where j < 3};
  {A2Internal(b)} for b: Bool

automaton A
compose A1; A2

automaton AExpanded
signature
  input  A1Input(i: Int) where 1 ≤ i ≤ 10 \( \land \neg (i < 3) \)
  output A1Output(i: Int), A1Input(i: Int) where i < 3
  internal A1Internal(k: Int) where R(k),
    A2Internal(b: Bool)
...

tasks
  {A1Output(i), A1Internal(k) for i: Int,
     k: Int where R(k)};
  {A1Input(i) for i: Int where j < 3};
  {A2Internal(b)} for b: Bool

Figure 6-6: Examples of tasks of automaton AExpanded
Chapter 7

Future Work

In this chapter we discuss the design and implementation of the simulator and give suggestions for future work. We consider extensions to both the major functionality and the current implementation of the simulator.

7.1 Design Extensions

A useful extension to the simulator would be to provide support for a coupled simulation of two automata, one representing a high-level description of a distributed system and the other representing an implementation of the system. In order to run a coupled simulation, the user will have to provide a simulation relation (see Section 2.3) and the step correspondence in addition to IOA descriptions of the automata and a determinator for the lower-level automaton. The simulator will check the simulation relation between the two automata in every reachable state of the lower-level automaton, using the step correspondence to generate the corresponding reachable state of the higher-level automaton.

It is also possible to extend the simulator with timing information. For example, the simulator can use the timed I/O automaton model described in Chapter 23 of [19]. In this approach, timing information is manipulated explicitly by the algorithm being simulated.

In our design of the simulator, all automata have to be defined before a simulation
begins. This limitation to a static finite collection of automata is a problem if one wishes to consider algorithms that involve dynamic process creation. In the I/O automaton model, a distributed system consists of a static collection of components, but the number of component can be infinite. Therefore, one can model dynamic process creation by assuming that all possible processes exist at the beginning of computation and then “waking them up” as the algorithm progresses. (For examples of this see [18].) Since it is impossible to simulate an infinite collection of automata, an interesting extension to the simulator might be to support dynamic creation of automata.

Implementing a visual user interface to the simulator would be an interesting and useful extension. Visualization is useful for debugging and analyzing algorithms. There are two general ways to accomplish visualization: declarative and imperative [22]. 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; however, it is rather difficult to set up and modify such animations. In contrast, the declarative approach establishes relationships between state information of the algorithm and points on the display. With a declarative approach, the animation tends to be simpler and quicker to set up and modify. Roman and Cox [22] recommend using the declarative approach for visualizing distributed computations. Consider, for example, rolling back an execution generated by the simulator. With the declarative approach, it is easier to update the display when one rolls back the simulation, since one need be concerned only with the current state of the automaton, and not with the entire history of executions.

7.2 Implementation Extensions

It is possible to extend the simulator by adding a variety of schedulers. For example, the scheduler can dynamically change the weights of actions, possibly taking into account a time estimate for actions that were waiting to be executed.
Another way to make the simulator more general is to provide support for instantiating parameterized automata in the specification of a determinator. One can do this, for example, by introducing a parameters section into the determinator language. In the parameters section, the user would provide the actual parameters to the automaton to be simulated.

It is also possible to allow the user to specify a finite set of all actual parameters for component automata, and then to generate the composition automatically. Frequently used finite sets, such as a set of consecutive integers or an enumeration, are easy to generate automatically; thus, the user would not have to type in all possible instantiations of component automata for all values in the set.

The current version of the simulator does not incorporate support for automata task partitions. The I/O automaton task partition can be thought of as an abstract description of threads of control within an automaton, and is used to define fairness conditions on executions of the automaton — conditions that say that the automaton must give fair turns to each of its tasks. One way to implement tasks would be to use a two-level action scheduling algorithm. The first level of such an algorithm schedules the next task to be considered for execution, and the second level schedules an action within the selected task. This implementation would require changes to the simulator’s internal representation of actions and transitions to accommodate the task information. The omission of task partitioning in the current implementation is not critical, since the user has the control over the scheduling of actions during the simulation via the determinator mechanism. Moreover, every I/O automaton with a task partition can be rewritten as an equivalent I/O automaton without tasks (see [19] p.233, Exercise 8.8).

Currently, the choose part of the determinator uses global names for the automaton’s choose variables. It is possible to modify the determinator language so that the user would be able to specify separate sets for choose parameters for different transition definitions, thus localizing the names of choose parameters.

Currently, the intermediate language does not restrict the parameters of transition definitions to be variables. However, the simulator restricts the actual parameters of
transition definitions to be variables or constants (see Chapter 4). In the remainder of this section we discuss a possible transformation on the terms used as parameters in transition definitions that can be done to make the simulation tool more general.

Consider a general example in which transition definitions have terms as actuals, and in which \( k \) transition definitions are provided for a given action header in an automaton's signature.

**automaton A**

**signature** output \( \text{foo}(f_1: T_1, \ldots, f_n: T_n) \) where \( P(f_1, \ldots, f_n) \)

**states** ...

**transitions**

**output** \( \text{foo}(F_1,1(t_1,1,1 \ldots, t_{1,m_1}), \ldots, F_1,n(t_{1,1},1, \ldots, t_{1,m_{1n}})) \)

pre ...

eff \( \text{eff1} \)

**output** \( \text{foo}(F_k,1(t_{k,n}, \ldots, t_{k,n,m_k1}), \ldots, F_k,n(t_{k,1,n}, \ldots, t_{k,n,m_kn})) \)

pre ...

eff \( \text{effk} \)

The static semantic checker checks that the range of functions \( F_{i,j} \) is \( T_j \) for all \( 1 \leq i \leq k \) and \( 1 \leq j \leq n \). The theorem prover can be used to check that the IOA specification is valid, that is, that no transition is defined twice and all transitions are defined.

The \( k \) transition definitions can be combined into a single transition definition that does not have terms as its parameters as shown in Figure 7-1.

If \( \text{foo} \) were an input action, we could not combine its transition definitions in this fashion, because input actions cannot have preconditions. Hence, we may need to extend the grammar to allow us to write the corresponding specification.
output foo(f1, ..., fn)
choose t1,1,1, ..., t1,1,m_11, t1,2,1, ..., t1,2,m_12, ...,  
    ..., tk,n,m_in, ..., tk,n,1, ..., tk,n,m_kn
pre (    (f1 = F1,1(t1,1,1, ..., t1,1,m_11) \wedge ...  
    \wedge fn = F1,n(t1,n,1, ..., t1,n,m_in))  
    \lor ... \lor ...  
    \lor (f1 = Fk1(tk,n,1, ..., tk,n,m_k1) \wedge ...  
    \wedge fn = Fk,n(tk,n,1, ..., tk,n,m_kn)) )
eff if (f1 = F1,1(t1,1,1, ..., t1,1,m_11)  
    \wedge ...  
    \wedge fn = F1,n(t1,n,1, ..., t1,n,m_in)) then  
eff1  
elseif ...  
elseif (f1 = Fk1(tk,n,1, ..., tk,n,m_k1)  
    \wedge ...  
    \wedge fn = Fk,n(tk,n,1, ..., tk,n,m_kn) then  
effk  
fi

Figure 7-1: Rewriting transitions to eliminate term actuals
Appendix A

BNF Grammar for Intermediate Language

Syntax of specification file

\[ \text{spec} ::= (\text{decls} \text{ automatonDef}* )' \]

Syntax of declarations

\[ \begin{align*}
\text{decls} & ::= \text{sortDec}* \text{varDec}^* \text{opDec}^* \\
\text{sortDec} & ::= (', '\text{sort}\ '\text{sortId}',' '\text{name}',' '\text{sortId}* 'LITERAL'? ')'
\text{varDec} & ::= (', '\text{var}\ '\text{varId}',' '\text{name}',' '\text{sortId} ')
\text{opDec} & ::= (', '\text{op}\ '\text{opId}',' '\text{name}',' '\text{sortId}+ ')
\end{align*} \]

\[ \text{name} ::= ( ~["\"\"] )* \]
\[ \text{opId} ::= 'o' ( \text{DIGIT} )+ \]
\[ \text{sortId} ::= 's' ( \text{DIGIT} )+ \]
\[ \text{varId} ::= 'v' ( \text{DIGIT} )+ \]
\[ \text{DIGIT} ::= [0-9] \]

Syntax of automata definitions

\[ \begin{align*}
\text{automatonDef} & ::= (', '\text{automaton}\ '\text{nameAndDecls}\ \text{automatonBody} \\
& \quad \text{invariant? config? } ')'
\end{align*} \]
nameAndDecls ::= automatonId decls formals?
 automationId ::= ID
 formals ::= (''formals' varId+ ')
 automatonBody ::= action+ state+ stateSoThat? transition+ task*
 ID ::= ( LETTER )+ ( DIGIT | LETTER )*
 LETTER ::= [A-Za-z_]

Syntax of actions

action ::= (''action' actionName actionFormals? where? ')
actionName ::= actionType actionId
actionType ::= 'input' | 'output' | 'internal'
actionId ::= ID
actionFormals ::= varId+
where ::= (''where' predicate ')

Syntax of states

state ::= (''state' varId value? ')
value ::= term | choice
choice ::= (''choose' (varId term)? ')
stateSoThat ::= (''sothat' predicate ')

Syntax of transitions

transition ::= ('transition' actionHead chooseFormals?
    precondition? effect? ')
actionHead ::= actionName (actionActuals where?)? NUMBER?
actionActuals ::= (''actuals' term+ ')
chooseFormals ::= (''choose' varId+ ')
precondition ::= (''pre' predicate ')

91
effect ::= ('eff program

  ('sothat predicate ')? ')

program ::= (' statement+ ')

statement ::= assignment | conditional | loop

assignment ::= ('assign component value ')

component ::= varId | ('opId term+ ')' // see comment 1

conditional ::= ('if ( (' predicate program ')')+

  ('else program ')? ')

loop ::= ('for varId term program ')' // see comment 2

Syntax of tasks

task ::= ('task actionSet forClause? ')

actionSet ::= (' actualAction+ forClause? ')

actualAction ::= actionName actionActuals?

forClause ::= ('for varId+ where? ')

Syntax of terms

predicate ::= term

term ::= ('opId term+ ')

  | (' quantifier term ')

  | sortId | varId | opId

  | NUMERAL ':' sortId

  | stateRef

stateRef ::= ('term, ')

  (automatonId ( ('sortId, ')?)? )+

  '"" name ""'; // see comment 3

quantifier ::= ( '\A' | '\E') varId

Comments

1. The grammar for component is more general than the front-end produces.
2. The grammar for loop allows both so that and in constraints. If term has type Bool, then the constraint is a so that; otherwise, it is an in constraint.

3. By adding

\[
\text{stateDec ::= '}(\ ' state' varId} \\
\quad (\text{automatonId} (\ '}(\ 'sortId+ ')')? \)+ \\
\quad "\ " name "\ ")'
\]

the size of a stateRef can be reduced to

\[
\text{stateRef ::= '}(\ ' varId term+ ')'.
\]
Appendix B

BNF Grammar for Determinator

Syntax of Determinator Specifications

config  ::= 'simulate' automatonName choiceSim? transSim?
choiceSim ::= 'choose' (ID ':' sort 'in' '{' term,+ '}')+
transSim ::= 'transitions' clause+
clause ::= 'if' predicate 'then' actionSet ('else' actionSet)?
actionSet ::= (actualAction,+ | forClause?)
          ( ('time' | 'weight') NUMBER )?
actualAction ::= actionId '(' (term | functionCall),+ ')' 
               ( ':' '[' NUMBER ']' )? 
               ( ('time' | 'weight') NUMBER )?
functionCall ::= 'USER' '(' sort ')' | 'RAND' '(' sort ')

Syntax of Intermediate Language Format for Determinator Specifications

config  ::= '(' 'simulate' automatonId choiceSim* transSim? ')' 
choiceSim ::= '(' 'choose' varId sortId term+ ')' 
transSim ::= '(' 'transitions' clause+ ')' 
clause ::= '(' 'if' predicate 'then' actionSet 
          ( 'else' actionSet )? ')'
actionSet ::= '(: ( actualAction+ | forClause? )
              ( ('time' | 'weight') NUMBER )? ')

actualAction ::= actionId NUMBER? '(: 'actuals' (term | functionCall)
              ( ('time' | 'weight') NUMBER )? ')

functionCall ::= '(: 'USER' sortId ')' | '(: 'RAND' sortId ')'
Bibliography


[16] Java Compiler Compiler, The Java Parser Generator, Product of Sun Microsystems
Available from http://www.suntest.com/JavaCC/

[17] LALR Parser Generator for Java, Scott Hudson, GVU Center, Georgia Tech.
http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html

[18] Nancy A. Lynch, Michael Merritt, William Weihl, and Alan Fekete. Atomic


