Verifiable Compilation of I/O Automata without Global Synchronization

by

Joshua A. Tauber

B.S., Computer Science (1991)
B.A., Government (1991)
Cornell University
S.M., Electrical Engineering and Computer Science (1996)
Massachusetts Institute of Technology

Submitted to the

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 16, 2005

Author .....
Department of Electrical Engineering and Computer Science
September 17, 2004

Certified by .....
Nancy A. Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

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

Verifiable Compilation of I/O Automata without Global Synchronization

by

Joshua A. Tauber

Submitted to the Department of Electrical Engineering and Computer Science
on September 17, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Part I of this thesis presents a strategy for compiling distributed systems specified in IOA into Java programs running on a group of networked workstations. IOA is a formal language for describing distributed systems as I/O automata. The translation works node-by-node, translating IOA programs into Java classes that communicate using the Message Passing Interface (MPI). The resulting system runs without any global synchronization. We prove that, subject to certain restrictions on the program to be compiled, assumptions on the correctness of hand-coded datatype implementations, and basic assumptions about the behavior of the network, the compilation method preserves safety properties of the IOA program in the generated Java code. We model the generated Java code itself as a threaded, low-level I/O automaton and use a refinement mapping to show that the external behavior of the system is preserved by the translation. The IOA compiler has been implemented at MIT as part of the IOA toolkit. The toolkit supports algorithm design, development, testing, and formal verification using automated tools.

The IOA language provides notations for defining both primitive and composite I/O automata. Part II of this thesis describes, both formally and with examples, the constraints on these definitions, the composability requirements for the components of a composite automaton, and the transformation a definition of a composite automaton into a definition of an equivalent primitive automaton.

Thesis Supervisor: Nancy A. Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

I use the "editorial we" throughout this dissertation not just to maintain formality, but also to acknowledge the years of work by the nearly two dozen people who have been involved in the IOA project. I collaborated with members of the project team at almost every step of the design, implementation, and testing of the IOA compiler and composer. I would like to thank them all for creating such a stimulating and pleasant environment in which to work.

When I joined the Theory of Distributed Systems group (TDS) in the Fall of 1997, the IOA project was already under way. Nancy Lynch and Steve Garland had drafted the IOA language and Steve had built the first version of the IOA parser. In our very first discussions, Nancy suggested that IOA might be used to build distributed systems.

Within a few months, Anna Chefter developed the first designs for the IOA simulator and the composer. Mandana Vaziri built a prototype IOA-to-Promela translation tool to connect the IOA toolset to the Spin model checker. Mandana also worked with Steve to develop the formal semantics for the IOA language. Nancy and I worked out the basic design of the IOA compiler at this time.

From the beginning, it was clear that resolving nondeterminism would be a big challenge in compiling the language. The same challenge faced Antonio Ramırez-Robredo as he extended Anna's design for the IOA simulator to handle pairs of automata. While I helped Mandana to develop the NAD transformation to eliminate nondeterminism from automata, Antonio took a different tack. He designed and implemented the first version of the NDR language for annotating automata with schedules. Antonio's elegant implementation of the simulator set the pattern that all other IOA tools, including the compiler, follow.

In the Spring of 1999 — at the end of his sophomore year, Michael Tsai joined the IOA project. Over the next three years, his dedication, his unfailing eye for design, and his prodigious capacity to assimilate, organize, and generate code advanced the the IOA compiler and the whole IOA project immeasurably. By the time Michael graduated, he had coded nearly the entire compiler. It is humbling to note just how long it took me to "finish it up" and hard to imagine how the compiler would have been built without him. It was a genuine pleasure to work with Michael.

The following year, Toh Ne Win took on the impressive task of learning the code bases for two separate projects — IOA and Daikon. He mastered both and worked with Mike Ernst to use each tool to improve the other. Together, Michael Tsai and Toh answered nearly every question I ever had about the toolkit code and suggested — and usually implemented — multiple solutions to every design challenge I raised.

Quickly, the size of the IOA project team more than doubled. Andrej Bogdanov connected IOA to the Larch Prover. Stan Funiak built the prototype IOA to TLA translator to support model checking. Atish Dev Nigam worked with Michael Tsai to support user customization of the compiler. Holly Reimers wrote the IL unparser that let backend tools regenerate IOA source code. The trio of

Laura Dean, Christine Karlovich, and Ezra Rosen began using the tools to implement and analyze classic distributed algorithms. Chris Luhrs took to analyzing graph algorithms and exercising the connection between IOA and Larch. I would like to thank everyone for making the summer of 2000 the memorable, challenging, collaborative, and truly fun experience it was.

In the fall of 2000, Laura took over work on the IOA simulator and worked with Toh to connect it to Daikon. In 2001, Dilsun Kırlı Kaynar became the fourth staff member on the project and the fifth person to take responsibility for the IOA simulator when she began her post-doc with TDS that fall. In turn, Edward Solovey extended the simulator to work with composite automata. Christine Robson contributed a tool to translate IOA for use with Uppaal.

When Steve and I began to build the composer tool in the winter of 2001 based on notations and mechanisms we had worked out with Nancy and Mandana, we thought the project would take a couple of weeks. We had no idea our design would turn out to be as complicated as it is or that the design process would stretch out to more than a year and a half. In that design process, Steve demonstrated endless patience, a tireless eye for detail, and expert craftsmanship. He carefully untangled the morass of variables, quantifiers, and predicates and he encouraged me to find the path through. We published the resulting design as MIT LCS Technical Report 959 [118]. That TR forms Part II of this dissertation. Throughout the composer design process, Dilsun provided a frequent sounding board. She suggested numerous and substantial clarifications in presentation.

In the fall of 2002, I began collaborating with Peter Musial on the implementation of the composer. Unfortunately, at that time, the design for the composer was only half baked. None the less, Peter took up the cause, volunteering to commute from UConn for weekly meetings. Many improvements to the design and implementation of the composer are due to Peter.

In the Spring of 2004, my attention was divided between writing the bulk of this document, finishing the composer, and finishing the compiler. Once again, I received the benefit of a wonderful collaboration. In this case, the newest member of the IOA project team, Panayiotis Mavrommatis, stepped in to dissect and extend the very innermost and tangled parts of the compiler that had been collecting dust for years. He was able to lay out the design choices so that even I could understand them. He proceeded to be the first to get the compiler up and running. When Panayiotis went home to Cyprus for the summer, we began a new fruitful collaboration with Chryssis Georgiou at the University of Cyprus. Panayiotis and Chryssis undertook a program of experimentation with the compiler that laid the ground work for the experimental evaluation that appears in this dissertation.

I would like to thank Nancy and Steve for years of guidance, support, and — not least — confidence. I learned much from trying to follow their sterling examples of scholarship. I want to thank Michael Ernst for always asking the hard questions and gamely attempting the futile task to injecting structure into my work habits. Thanks go to Martin Rinard for his help in setting the parameters of this project during his Advanced Compilers seminar way back in the Fall of 1998.

I would also like to thank all the members of TDS not involved with IOA for creating a wonderful atmosphere in which to work. Thank you Rui Fan, Seth Gilbert, Roger Khazan, Carl Livadas, Victor Luchangco, Sayan Mitra, Tina Nolte, Roberto de Prisco, Paul Attie, Idit Keidar, and Alex Shvartsman. I especially thank Joanne Talbot Hanley for years of making things happen and for all the pistachios.

I would like to thank those who provided the little daily distractions that smoothed my way through life at MIT. Chris Peikert, David Liben-Nowell, Grant Wang, and the rest of the LCS New York Times Crossword Puzzle Team made many lunch times disappear. Jonathan Kelner provided endless entertainment as we discussed the moral superiority of our respective professional baseball teams. Thank you Victor for hours of entertaining argument — whatever the topic.

I would like to thank the MIT Writing Center and, in particular, its director/miracle worker Steve Strang. I cannot even contemplate how my dissertation would have been written without your patient support. Thank you Mark Greenberg for being a steady reality check. Finally, I thank my friends and family for years of support, caring, and understanding.

This dissertation brought to you by the letters M, I, and T, by the number 42, and by

- Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership,
- DARPA through the Office of Naval Research under contract number N66001-99-2-891702,
- DARPA/AFOSR MURI Awards F49620-02-1-0325 and SA2796PO 1-0000243658
- DARPA contracts F19628-95-C-0118 and F33615-01-C-1896,
- Air Force Aerospace Research-OSR Contracts F49620-00-1-0097, F49620-97-1-0337, and FA9550-04-1-0121,
- NTT Contract MIT9904-12 ,
- Draper Contract DL-H-543107,
- NSF Grants ACI-9876931, CCR-9909114, CCR-9804665, CCR-0326277 and CCR-0121277, and
- NSF-Texas Engineering Experiment Station grant 64961-CS.

Contents

List of Figures

List of Tables

Part I

IOA Compiler

Chapter 1

Introduction

Computer /nm./: a device designed to speed and automate errors.

— Anonymous

1.1 Motivation

Distributed systems are increasingly prevalent and increasingly complicated. Complex and interconnected distributed services are being rolled out daily in areas as diverse as view-oriented group communication, mobile IP routing, computer-supported cooperative work (CSCW), and transportation control systems. Many such systems are useful tools for the public as well as objects of study for computer scientists. Unfortunately, the concurrent nature of these systems intertwined with their scale and complexity make building and reasoning about them notoriously difficult. Concurrency is inherent in any computation distributed over a collection of computing elements. That concurrency can result in subtle interactions among those components that cause a distributed system to behave in ways unintended by its designers. The state of the art for building reliable distributed systems is to cycle through phases of designing, implementing, and extensively testing the system. However, the best efforts of system builders have failed to eliminate key logical errors and ambiguities in prominent, widely deployed systems [5, 39, 54, 55, 58].

At the same time researchers have developed a variety of formal models for distributed computation [84, 85, 59, 90, 72]. Formal methods and modeling can greatly aid system builders in understanding, analyzing, and designing distributed systems. In a mathematical and state-machine-based approach, one describes systems in a structured way, viewing them (orthogonally) as parallel compositions of interacting components, and as sequences of descriptions at different levels of ab-

straction. These formalisms have been used to successfully model and verify a wide variety of distributed systems and algorithms and to express and prove many impossibility results. Many of the modeling and verification techniques based on state machines are highly stylized. Recurring patterns of successful proofs using simple formal models have led a variety of researchers to explore using formal languages and automated tools, such as automated proof assistants, model checkers, simulators, and even compilers, to work with these models [45, 2, 106, 27, 35, 123]. Until now, it has not been possible to apply these tools to the same formal expression of an algorithm both to prove its correctness and to build a distributed implementation of it.

In this dissertation, we describe a method to produce verified running code for distributed systems. We present a strategy for compiling formal models of distributed algorithms into Java programs running on a collection of networked workstations. Furthermore, we prove that, under precisely stated conditions, the compilation method preserves the safety properties of the program in the running system. We model distributed systems as I/O automata. I/O automata provide a simple mathematical basis and a rich set of proof techniques for formally modeling, understanding, and verifying distributed systems. We express I/O automata using the IOA language. IOA is a formal language for describing I/O automata that serves both as a formal specification language and as a programming language [43].

The IOA compiler we present in this dissertation forms part of the IOA toolkit. The IOA toolkit supports algorithm design, development, testing, and formal verification using automated tools. The toolkit connects I/O automata together with both lightweight (syntax checkers, simulators, model checkers) and heavyweight (theorem provers) tools. The IOA toolkit enables programmers to write their specifications at a high level of abstraction, use tools to validate the specification, successively refine the specification to a low-level design, and then automatically translate the design into code that runs on a collection of workstations communicating via standard networking protocols. A major contribution of this work is that the compiler overcomes the existing disconnect between correctness claims for formal specifications and actual system implementations by allowing the same IOA program to be both verified with automated tools and compiled into a running distributed system.

1.2 Challenges

The design and implementation of the IOA compiler required overcoming a number of key challenges. Many of these challenges arise from the differences between characteristics of specifications that are easiest to prove correct and characteristics of programs that are easiest to run. In this section, we introduce these challenges and our approaches to addressing them.

1.2.1 Structuring programs for compilation

The first major challenge our work addresses is how to create a system with the correct externally-visible behavior of the system without using any synchronization between processes running on different machines. We achieve this goal by matching the formal specification of the distributed system to the target architecture of running systems. That is, we restrict the form of the IOA programs admissible for compilation. We require the programmer rather than the compiler to decide on the distribution of computation. Specifically, we require the programs submitted for compilation to be structured in a *node-channel* form that reflects the message-passing architecture of the collection of networked workstations that is the target of the compiler. Had we been compiling to a shared-memory system, we would have required the IOA programs to be written in that style. (IOA itself is flexible enough to describe systems architected as completely centralized designs, shared memory implementations, or message passing arrangements.) Compilation then proceeds on a node-by-node basis. That is, the code for each kind of node is compiled separately. In fact, each node in the system may run entirely specialized code. By requiring the programmer to match the system design to the target language, hardware, and system services before attempting to compile the program, we are able to generate verifiably correct code without any synchronization between processes running on different machines.

1.2.2 Connecting programs to system services

Of course, IOA systems do not run in isolation. IOA programs use external services such as communication networks and console inputs. The IOA compiler generates only the specialized code necessary to implement an algorithm at each node in the system. At runtime, each node connects to the external system services it uses. In our prototype the compilation target is a system in which each host runs a Java interpreter and communicates via a small subset of the Message Passing Interface (MPI)) [41, 4]. A second major challenge our work addresses is to create both correctness proofs about algorithms that connect to such services and to produce correct code that uses such external, preexisting services.

Our approach to external system services is to bring them into the formal model by creating automata that make explicit all our assumptions both about the interfaces to those service and about all the externally visible behaviors of those services. For example, we introduce an automaton that models a subset of MPI in Chapter 4. IOA programmers use these models to prove the correctness of systems dependent on the external services. Distributed system designers can produce conditional proofs of correctness for an entire system even though nodes communicate via MPI.

The correctness of the systems accessing other external system services can be verified by following the same general approach. We can model an external service by writing an IOA specification.

Subsequently, proofs of correctness about programs that use the service must consider the entire system including the modeled service. Such proofs are conditioned on the assumption that the external service behaves as described in our model.

However, requiring the programmer to write code specifically to model the particulars of procedure calls to specific external services is more restrictive than necessary. Writing programs at such a low level complicates system designs unnecessarily and, thus, makes verifying the correctness of systems harder. We avoid this complexity by specifying abstract services designers want to use (*e.g.*, point-to-point, reliable, FIFO channels) and then implementing these abstract services by combining our model of the external service with auxiliary mediator automata. We then verify that this design implements the desired abstract service.

Such a proof does involve just the sort of details about low-level interactions with the external service we wish to avoid. However this proof need only be performed once to verify the compiler design. Programmers may then assume the existence of the simpler abstract service in any proofs about their programs.

Thus, our strategy for verifying access to an external service is a four step process. First model the external service as an IOA automaton. Second, identify the desired abstract service programmers would like to use and specify that abstract service as an IOA automaton. Third, write mediator automata such that the composition of the mediator automata and the external service automaton implements the abstract service automaton. Fourth, prove that implementation relationship.

1.2.3 Modeling procedure calls

The above design for connecting to system services raises new challenges. One particularly tricky aspect of such proofs is modeling the interfaces to services correctly. IOA itself has no notion of procedure call *per se*. There are two reasons to avoid procedure calls in a specification language. First, procedure call stacks complicate the state of programs and, therefore, proofs of correctness about them. Second, when one considers procedure calls as the interface between interacting components in a concurrent setting, the procedure call and the procedure return should often be considered to be two separately visible events across the interface.

The Java interface to an external service is defined in terms of method invocations (procedure calls). In our models of these services, we carefully treat method invocations and method returns as distinct behaviors of the external service. When procedure calls may block, we describe handshake protocols to model such blocking.

1.2.4 Composing automata

The auxiliary mediator automata created to implement abstract system services must be combined with the source automaton prior to compilation. We compose these automata to form a single au-

tomaton that describes all the computation local to a single node in the system. (Composition across nodes is performed as part of proofs concerning the behavior of an entire distributed system but not as part of compiling such a system.) We have designed and implemented a tool to compose automata automatically. Part II of this dissertation precisely defines syntactic manipulations that transform an automaton that is described as a combination of component automata into a single equivalent primitive automaton. As part of this definition we have defined a desugared core IOA language and shown syntactic transformations to produce desugared automata from arbitrary primitive automata.

1.2.5 Resolving nondeterminism

The IOA language is inherently nondeterministic. Translating programs written in IOA into an imperative language like Java requires resolving all nondeterministic choices. This process of resolving choices is called scheduling an automaton. Developing a method to schedule automata was the largest conceptual challenge in the initial design of an IOA compiler. In general, it is computationally infeasible to schedule IOA programs automatically. Instead, we augment IOA with nondeterminism resolution (NDR) constructs that allow programmers to schedule automata directly and safely.

1.2.6 Implementing datatypes

Datatypes used in IOA programs are described formally by axiomatic descriptions in first-order logic. While such specifications provide sound bases for proofs, it is not easy to translate them automatically into an imperative language such as Java. However, the IOA framework focuses on correctness of the concurrent, interactive aspects of programs rather than of the sequential aspects. Therefore we are not especially concerned with establishing the correctness of datatype implementations. (Standard techniques of sequential program verification may be applied to attempt such correctness proofs.) Therefore, each IOA datatype is implemented by a hand-coded Java class. A library of such classes for the standard IOA datatypes is included in the compiler. Each IOA datatype and operator is matched with its Java implementation class using a datatype registry written by Michael Tsai and Toh Ne Win, extending an original design by Antonio Ramırez-Robredo [119, 121, 102].

1.3 Correctness

1.3.1 Abstract Channel Correctness

As suggested above, even though IOA programs use MPI as the network service, we allow programmers to assume the existence of point-to-point, reliable, FIFO channels when designing distributed systems. We are able to provide programmers this convenience by following our four step strategy for connecting to external services. First, we model MPI as an IOA automaton. Second, we specify

the desired semantics of a point-to-point, reliable, FIFO channel as an IOA automaton. Third, we define mediator automata and compose them with our MPI model. Fourth, we prove that this composite channel definition implements the desired abstract channel automaton.

1.3.2 Compiler Correctness

Since the goal of building the IOA compiler is to bridge the gap between formal models and running systems, we need to ensure that the compilation process maintains the semantics of the source IOA program in the emitted target code. We prove that under precisely stated conditions the compilation method preserves the safety properties of the source IOA program in the target Java code.

To construct this proof we model the emitted Java code as another IOA automaton. As with any useful model, our model of the emitted Java code abstracts away irrelevant detail while modeling key characteristics. In the case of the emitted Java code the key characteristics we model are concerned with multithreading, the granularity of atomic steps, and control flow. Our model of Java programs is multithreaded, and threads may share variables and may synchronize using locks. Our IOA automaton model of Java programs can take arbitrarily small "micro-steps" that may be interleaved across threads and nodes.

We model the compilation process as a syntactic transformation on IOA automata. Abstractly, a source IOA automaton is "compiled" by mapping it into a micro-step automaton in a precisely defined way. We show the correctness of the compilation strategy by proving that any micro-step automaton resulting from such a transformation maintains all the safety properties of the source automaton from which it was generated. We construct this proof by demonstrating a refinement mapping from a target automaton to the source automaton from which it was compiled.

This proof is based on the assumptions that our model of network behavior is accurate, that our hand-coded datatype library correctly implements its semantic specification, and that the NDR annotations of the source automaton produce valid values. Moreover, we assume a technical restriction that the source automaton is designed so that its safety properties hold even when inputs to any node in the distributed system are delayed. Our current model of network behavior does not allow for failures.

This result is best viewed as a correctness condition for the compiler. Consider a source automaton and the micro-step automaton our precisely described syntactic transformation generates from it. The IOA compiler is correct if that the micro-step automaton accurately models the Java code emitted by the compilation of the source automaton.

1.4 Performance

We have performed three case studies to evaluate our implementation of the IOA compiler. For each, we have written an IOA automaton to implement a distributed algorithm from the literature. The three algorithms we have used are LCR leader election, computation of a spanning tree, and repeated broadcast/convergecast over a computed spanning tree. Even with an incomplete prototype of the compiler, an MIT undergraduate working at the University of Cyprus has been able to translate the latter two algorithms from the literature into running code in a matter of 4–6 hours [88].

We have compiled these algorithms and measured their performance. Our experimental testbed consists of a collection of ten networked workstations. In our limited experience, we have not observed any scaling overhead from the compilation process. For example, the runtime of LCR expands linearly with the number of participating nodes, just as expected. On the other hand, we observed that the performance of our current implementation degrades as the size of the state of the automaton increases. For example, the runtime of the broadcast algorithm expanded quadratically with the number of messages sent. We suggest a solution for this problem.

1.5 Overview

Part I of this dissertation proceeds as follows. Chapter 2 introduces the input/output automaton model, the IOA language, and the IOA toolkit, and places this work in the context of related tools-based approaches to formal methods. Chapter 3 describes the form an IOA program must have to be admissible for compilation. Chapter 4 describes the MPI communication service used by compiled IOA programs, details the semantics we assume about that service, and shows formally how to achieve an abstract channel interface with simple semantics using that more complex service. The chapter also presents an argument for the correctness of our abstract channel implementation. Chapter 5 discusses annotations a programmer adds to an IOA program to resolve its inherent nondeterminism. Chapter 6 describes the translation process. Chapter 7 presents an argument for the correctness of the compilation method. Chapter 8 presents our experimental evaluation of the compiler. Part II describing the design of the composer encompasses Chapters 9– 17. Chapter 18 summarizes both parts of this dissertation, assesses its impact, and suggests directions for future work.

Chapter 2

IOA Language and Toolkit

If a language doesn't affect the way you think about programming, it's not worth knowing.

— Alan J. Perlis [99]

2.1 Input/Output Automata

I/O automata provide a simple mathematical basis for understanding distributed systems [84, 85]. I/O automata model the behavior of systems of interacting components. Complex systems are decomposed into simpler pieces whose structure can be understood using levels of abstraction and, orthogonally, parallel composition.

An *I/O automaton* is a labeled state transition system. It consists of a (possibly infinite) set of *states* (including a nonempty subset of *start states*); a set of *actions* (classified as input, output, or internal); and a *transition relation*, consisting of a set of (state, action, state) triples (*transitions* specifying the effects of the automaton's actions).[1] An action π is *enabled* in state s if there is some triple (s, π, s') in the transition relation of the automaton. Input actions are required to be enabled in all states. We call the internal and output actions the *locally-controlled* actions.

2.1.1 Execution of I/O Automata

The operation of an I/O automaton is described by its *executions*, which are alternating sequences of states and actions. The externally visible behavior occurring in executions constitutes its *traces* (sequences of input and output actions). The idea is that actions describe atomic steps. While two (or many) actions may be enabled in a given state, the automaton performs only one transition at

[1] We omit discussion of *tasks*, which are sets of non-input actions.

a time. If a second action remains enabled in the state of the automaton after the first transition, it may then occur. Thus, even though both actions were simultaneously enabled, one will be ordered before the other in any single execution of the automaton.

I/O automata admit a *parallel composition* operator, which allows an output action of one automaton to be performed together with input actions in other automata; this operator respects the trace semantics. The result of applying the composition operator to a collection of compatible automata is a new automaton semantically equivalent to the original collection. The execution of a composition of interacting automata is also described with a global sequence of actions. That is, the execution of the composition of a collection of automata is a *single* alternating sequence of states and actions. Thus, the execution of a concurrent system is described sequentially. Furthermore, even though the *enabling* of an action is determined only by examining the state of its automaton and even though the *effect* of that action is localized to the state of that single automaton, the *scheduling* of the action is performed globally over the whole collection.

The I/O automaton model is inherently nondeterministic. In any given state of an automaton (or collection of automata), one, none, or many (possible infinitely many) actions may be enabled. As a result, there may be many valid executions of an automaton.

2.1.2 Proof Techniques

The I/O automaton model supports a rich set of proof techniques. *Invariant assertion* techniques are used to prove that properties of automata are true in all reachable states. (These date back at least to Owicki and Gries [97].)

One automaton is said to *implement* another if all of its traces are also traces of the other automaton. Pairs of automata can be related using various forms of simulation relations. A simulation relation is a mapping between the states of two automata that is maintained in all reachable states while preserving the external behavior of the automata. To prove a relation is a simulation relation, one demonstrates a correspondence between states and a *step correspondence* between the two automata, that is, one shows that for every step (state transition) of the implementation automaton there is an equivalent (possibly empty) sequence of steps that the specification automaton can take that will maintain the simulation relation [69, 83, 29]. Formally, a binary relation f over the states of two automata A and B, is a *simulation relation* if

1. for all states s_0 in the start states of A there is a start u_0 of in the start states of B such that $(s_0, u_0) \in f$ and

2. if s is a reachable state of A, u is a reachable state of B, $(s, u) \in f$, and (s, π, s') is a transition of A, then there is an execution fragment α of B starting with u and ending with some u' such that $(s', u') \in f$ and α has the same trace as π.

Demonstrating a simulation relation between two automata shows that one automaton implements the other. If the simulation relation is a function, we say it is a *refinement mapping*. A relation h between the states of two automata A and B is a *history relation* from A to B if h is a simulation relation from A to B and h^{-1} is a refinement from B to A. A succinct explanation of the model and many of its proof techniques appears in Chapter 8 of [81]. Simulation and history relations are thoroughly discussed in [83].

2.1.3 Using I/O automata

To use I/O automata, one typically begins by describing a distributed system as a global, high-level application or service. If possible, one uses a single, centralized I/O automaton to capture the most general description of the externally discernible behavior of the system. For example, a bank can be described as a set of accounts with owners and balances into which deposits can be made and from which withdrawals can be taken. To be as general as possible, one uses nondeterministic choices whenever possible. Then one describess key properties of the system using invariants. For example, one might assert that no balance may ever be negative and that the total amount of funds in the bank is always equal to the total deposits made minus the total withdrawls made.

A process of *successive refinement* then follows to describe the system as made up of lower-level services. These lower-level automata may be simpler to understand (individually), a more realistic depiction of a real system, or a model of an existing system we wish to use. Two orthogonal methods of refinement apply. First, *levels of abstraction* may be used to define interfaces between low-level services and high-level applications. So, a bank may depend on a secure, low-level, wire transfer service to move money around. A distributed bank branches application can be described on top of that wire transfer service. Second, one can apply *parallel decomposition* to describe a service or application as a collection of components. Thus, the wire transfer service may be made up of a collection of computing nodes and communication networks. In either case, the result is a set of lower-level I/O automata that are intended to implement the previously specified high-level service.

Having refined the high-level, global system specification into a low-level distributed system description, one wishes to prove that the low-level description implements the high-level specification. Doing so shows that all behaviors of the low-level system could be interpreted as valid behaviors of the specification. So, for example, no combination of accepted deposits and withdrawals at any set of branches can ever cause a balance to become negative. To perform this proof, one applies the composition operator to the various pieces of the low-level, distributed system specification. To complete the proof, one then demonstrates a simulation relation between the resulting automaton and the high-level, global system specification.

2.2 Successes with the I/O Automaton Model

I/O automata have been used to successfully model and verify a wide variety of distributed systems and algorithms and to express and prove several impossibility results. Examples include [81, 82, 57, 13, 21, 39, 40, 26, 110, 111]. The model was developed for reasoning about theoretical distributed algorithms but has since been applied to many practical services. For example, I/O automata have been applied to distributed shared memory [39, 37, 38], group communication [40, 26, 32, 58], and standard networking [111, 110, 112]. The resulting expositions and proofs have resulted from a structured, rigorous approach that has resolved ambiguities and uncovered errors. Logical errors have been found in algorithms underlying Orca [5] and Ensemble [54, 55] while unexpected behavior was found in T/TCP [14].

2.3 Related Models

These experiences applying the model highlight several key features useful for this style of work. The external behavior of automata are based on simple linear traces. Composition is based on synchronized external actions. Levels of abstraction are easily described with successive levels related by inclusion of trace sets. However, the I/O automaton model is just one of a variety of mathematical models developed for specifying and verifying distributed systems. A number of models describe systems as automata performing transitions with preconditions and effects (*i.e.*, guarded commands). The effects can be described operationally (by an imperative program) or axiomatically (by predicates that relate pre-states to post-states). I/O automata can be described by either or both methods as needed.

Lamport's TLA [71] describes the effects of transitions using the latter assertional style. TLA describes automata as constraints on the universe of all possible outcomes and composition of automata simply as the conjunction of such constraints. Thus, there is no clear analogy to traces as descriptions of external behavior. While Chandy and Misra's UNITY language [17] and Manna and Pnueli's language SPL [86] are operational in style, their automata combine via shared variables rather than shared actions. Thus, neither model uses sets of traces as its notion of external behavior. All three of these languages have been used to prove both safety and liveness properties using temporal logic.

Process algebras also use automata models. CSP [59] and CCS [90] compose by synchronizing external actions. However, the process algebra proof style is quite different from that used with I/O automata. Concurrent systems are built from single-step processes using algebraic expressions and operators. Proofs consist of the application of a rich set of inference rules to algebraic expressions denoting processes and tend to emphasize the equivalence of expressions.

A variety of other methods have been used for formally specifying distributed systems. For

example, see work by Harel [53], Meseguer [89], and Ostroff [96]. Estelle [62] exemplifies "Formal Description Techniques (FDT)." FDTs are high-level, highly-expressive programming languages with formal semantics. Estelle's semantics are based on a guarded command style automaton model [31]. While the semantics are carefully defined, proofs are not generally done with these systems, possibly due to the complications of the expressive semantics.

2.4 Related Tool-Based Work

Many of the modeling and verification techniques based on state machines are highly stylized. The properties researchers choose to identify and the structure of the proofs they use to verify these properties follow simple, common patterns. This success in combining simple, formal models with simple, recurring patterns of proofs has led a variety of researchers to explore using automated tools to work with these models. These computerized assistants have relieved researchers of some of the burden of rote and repetitive tasks associated with these models and the associated proofs. These tools range from verification-oriented tools such as automated proof assistants and model checkers to more design-oriented tools such as simulators and even compilers. They have been used to increase the level of detail and reusability in the produced proofs, to find errors in existing designs, to explore design spaces, and to connect specifications with produced code. However, no system has yet combined all these techniques for a single model.

A number of tools have been based on the CSP model [59]. The semantics of the Occam parallel computation language is defined in CSP [1, 2]. While there are a number of Occam compilers that target the Transputer architecture we have found no evidence of verification tools for Occam programs.

Formal Systems, Ltd., has developed a machine-readable language for CSP that is accepted by a number of tools. The FDR model checker allows the checking of a wide range of general safety and liveness properties of CSP models [106]. The ProBE tool enables the user to "browse" a CSP process by following events that lead from one state of the process to another. The user controls the resolution of non-determinism and the choice of actions.

Cleaveland *et al.* have developed a series of tools based on the CCS process algebra [90]. The Concurrency Workbench [28] and its successor the Concurrency Factory [27] are toolkits for the analysis of finite-state concurrent systems specified as CCS expressions. They include support for verification, simulation, and compilation. A model checking tool supports verifying bisimulations. A compilation tool translates specifications into Facile code.

Lamport developed TLA+[72] as a formal language for describing TLA automata in a modular fashion. With Engberg and Grønning, Lamport developed the TLP [35] theorem prover based on the Larch Prover. Yu and Manolios collaborated with Lamport to develop the TLC model checker [123]

for TLA automata specified in a subset of TLA+. TLC has been used to find errors in the cache coherence protocol for a Compaq multiprocessor. Lamport has no desire to generate code from TLA+ [73]. Kalvala has formulated TLA for Isabelle [64].

Bjørner *et al.* have developed the Stanford Prover (STeP) for verifying SPL automata [10, 9]. STeP combines model checking with deductive methods to allow the verification of a broad class of systems, including programs with infinite data domains. We do not know of any STeP work on code generation.

2.5 Related I/O Automata-Based Tools

2.5.1 Theorem Provers

A number of researchers have revisited proofs of algorithms previously described in the distributed systems literature using automated theorem proving assistants such as the Larch Prover, PVS, and Isabelle [44, 108, 98].

At MIT, researchers have encoded I/O automata theory in the Larch Shared Language (LSL) for use by the Larch Prover (LP). Søgaard-Andersen *et al.* developed the method and applied it to connection management protocol examples [113]. Söylemez used Larch to verify the timing properties of MMT automata [115]. Luchangco *et al.* extended the method to timed I/O automata and applied it to mutual exclusion and leader election algorithms [80, 79]. Probably the most complicated LSL-based proof done in this style was accomplished by Petrov *et al.* in verifying the Bounded Concurrent Timestamp algorithm of Dolev and Shavit [100]. In all these proofs, the researchers specified the algorithms of interest as I/O automata. The researchers then encoded each automaton and the relevant invariants and simulation relations in LSL for input to LP.

Müller, Nipkow, and Slind have encoded I/O automata theory using temporal logic for use with the Isabelle theorem prover [95, 92, 91]. Müller has used the system to derive and verify many theorems about I/O automata including standard proof methods. Müller has connected the system to two model checkers, translating I/O automata embedded in Isabelle into inputs for the the STeP and μcke model checkers [8, 6, 7]. He uses abstractions from finite to infinite automata to use model checkers to prove general properties. The model checker is used to verify the properties of the finite state automata while the theorem prover is used to verify the abstraction. Each automaton must be hand encoded directly in the Isabelle logic.

Vaandrager *et al.* have encoded I/O automata theory for the PVS theorem prover [56, 34]. As with previous efforts, the authors hand encoded individual automata in the input logic of the prover.

Archer and Heitmeyer also encoded I/O automata theory for PVS [3, 104]. They used PVS's support for user-defined strategies to create a special-purpose interface to PVS. The tool TAME (Timed Automata Modeling Environment) provides an interface to the prover PVS to simplify specifying

and proving properties of automata models (including both timed and untimed I/O automata). TAME aims to allow a software developer who has basic knowledge of standard logic, and can do hand proofs, to use PVS to represent and to prove properties about an automaton model without first becoming a PVS expert. In addition, TAME produces mechanical proof transcripts that largely correspond to the structure of Lamport-style hand proofs [74, 70].

2.5.2 Simulators and Code Generators

Goldman's Spectrum System introduced a formally-defined, purely operational programming language for describing I/O automata [47, 49]. He was able to execute this language in a single machine simulator. He did not connect the language to any other tools. However, he suggested a strategy for distributed simulation using expensive global synchronizations. More recently, Goldman's Programmers' Playground also provides a communication library with formal semantics expressed in terms of I/O automata [50].

Cheiner and Shvartsman experimented with methods for generating code from I/O automaton descriptions [23, 24, 25]. They selected a particular distributed algorithm from the literature (the Eventually Serializable Data Service of Luchangco *et al.* [37]) and generated by hand an executable, distributed implementation in C++ communicating via the Message Passing Interface (MPI [41]). They describe a generalized method for generating code for I/O automata described by operational pseudocode. Unfortunately, the general implementation strategy described uses costly reservation-based synchronization methods to avoid deadlock and a probabilistic, exponential back-off to avoid livelock in the reservation system itself. For certain automata, they are able to optimize this reservation system. Their methods do not rely on a formal language to describe I/O automata and have no direct connection to any verification support.

2.6 IOA Language

The history of success using I/O automata to analyze and verify complex distributed systems and the various efforts exploring tool-based verification methods clearly indicates the importance of the techniques being developed. In the previous modeling work, I/O automata are described using *pseudocode*. The use of pseudocode simplifies and clarifies a naïve application of the strictly mathematical set notation used to define the I/O automaton model by introducing programming style notations. For example, states are represented by state variables rather than just by members of an unstructured set; state transitions are described in precondition-effect style, rather than as state-action-state triples.

To promote wider application of the I/O automata-based techniques and to support the application of automated verification tools to I/O automata, Garland and Lynch introduced the *IOA*

language [43]. IOA is a formal language for describing I/O automata and their properties. IOA serves as both a formal specification language and a programming language. I/O automata described in IOA may be considered either specifications or programs. In either case, IOA yields precise, direct descriptions of I/O automata constructs. As in the pseudocode style that inspired the language, states in IOA are represented by the values of variables. IOA transitions are described in precondition-effect (or guarded-command) style. The precondition is a predicate on the state of the automaton and the parameters of the transition that must hold whenever the transition executes. The effects clause specifies the result of executing the transition.

Since the language is intended to serve both as a specification language and programming language, it supports both axiomatic and operational descriptions of programming constructs. Thus state changes can be described through imperative programming constructs like variable assignments and simple, bounded loops or by declarative predicate assertions restricting the relation of the post-state to the pre-state.

The language also directly reflects the nondeterministic nature of the I/O automaton model. Rather than add a few constructs for concurrency and interaction onto a basically sequential language, IOA is concurrent from the ground up. One or many transitions may be enabled at any time. However, only one is executed at a time. The selection of which enabled action to execute is the source of *implicit nondeterminism* in the language. The **choose** operator provides *explicit nondeterminism* in selecting values from (possibly infinite) sets. These two types of nondeterminism are derived directly from the underlying model. The first reflects the fact that many actions may be enabled in any state. The second reflects the fact that a state-action pair (s, π) may *not* uniquely determine the following state s' in a transition relation.

2.7 Example: LCR Leader Election

We illustrate IOA by describing the Le Lann-Chang-Roberts (LCR) leader election algorithm as a composition of process and channel automata [75, 18].

In this algorithm, a finite set of processes arranged in a ring elect a leader by communicating asynchronously. The algorithm works as follows. Each process sends its name to its right neighbor. When a process receives a name, it compares it to its own. If the received name is greater than its own, the process transmits the received name to the right; otherwise the process discards it. If a process receives its own name, that name must have traveled all the way around the ring, and the process can declare itself the leader.

Figure 2.1 shows a `LCRChannel` automaton describing communication channels by which processes can send messages.[2] This automaton represents a reliable communication channel, which neither

[2]Nothing about this channel definition is specific to the LCR example. We qualify the name only to distinguish the example from other channel automata introduced in Chapters 4 and 10.

```
automaton LCRChannel(i, j: Int)

  signature
    input SEND(m: Int, const i, const j)
    output RECEIVE(m: Int, const i, const j)

  states
    messages: Seq[Int] := {}

  transitions
    input SEND(m, i, j)
      eff messages := messages ⊢ m
    output RECEIVE(m, i, j)
      pre messages ≠ {} ∧ m = head(messages)
      eff messages := tail(messages)
```

Figure 2.1: Reliable FIFO `LCRChannel` automaton

loses nor reorders messages in transit. The automaton is parameterized by the values, `i` and `j`, which represent the indices of processes that use the channel for communication. The signature consists of input actions, `send(m, i, j)`, and output actions, `receive(m, i, j)`, one for each value of `m`. The keyword **const** in the signature indicates that `i` and `j` are terms (not variable declarations) whose values are fixed by the values of the automaton's parameters. The state of the automaton `LCRChannel` consists of a `buffer`, which is a sequence of messages (*i.e.*, an element of type `Seq[Int]`) initialized to the empty sequence `{}`. The operators on sequences used are: `{}` (the empty sequence), $\vdash$ (append), `head` (the first element of the sequence), and `tail` (the rest of the sequence). The input action `SEND(m, i, j)` appends `m` to `buffer`. The output action `RECEIVE(m, i, j)` is enabled when `buffer` is not empty and has the message `m` at its head. The effect of this action is to remove the head element from `buffer`.

Figure 2.2 describes a participating LCR process, which is parameterized by the name of the process and the number of processes participating in the election. The **type** declaration on the first two lines of Figure 2.2 declares `Status` to be an enumeration of the values `idle`, `voting`, `elected`, and `announced`. The automaton `LCRProcess` has two state variables: `pending` is a multiset of integers and `status` has type `Status`. Initially, `pending` is set to contain the name of the process `i`, and `status` is set to `idle`. The input action `vote` sets `status` to `voting` to indicate that an election has begun. The input action `receive(m, const mod(i-1, ringSize), const i)`, may result in three different transitions depending on how the message `m` received from the `LCRProcess` automaton to the left of automaton `i` compares with the name of automaton `i`. These transitions are described in three separate transition definitions; they could just as well have been described in a single definition using a conditional statement. The value of the first parameter of `receive` is constrained by **where** clauses in the first two transition definitions and is fixed in the third. The parameter `j` in each of these transition definitions is constrained to equal $\texttt{i} - 1 \mod \texttt{ringSize}$ by the action signature. The

```
type Status = enumeration of idle, voting, elected, announced

automaton LCRProcess(i, ringSize, name: Int)
  signature
    input vote(const i)
    input RECEIVE(m: Int, const mod(i-1, ringSize), const i)
    output SEND(m: Int, const i, const mod(i+1, ringSize))
    output leader(const i)

  states
    pending: Mset[Int] := {name},
    status: Status := idle

  transitions
    input vote(i)
      eff status := voting
    input RECEIVE(m, j, i) where m > name
      eff pending := insert(m, pending)
    input RECEIVE(m, j, i) where m < name
    input RECEIVE(name, j, i)
      eff status := elected
    output SEND(m, i, j)
      pre status ≠ idle ∧ m ∈ pending
      eff pending := delete(m, pending)
    output leader(i)
      pre status = elected
      eff status := announced
```

Figure 2.2: Algorithm automaton `LCRProcess`

automaton has two kinds of output actions: `send(m, i, mod(i+1,ringSize))`, which sends a message in `pending` to the `LCRProcess` automaton to the right, and `leader(i)`, which announces successful election.

```
automaton LCR
  components
    P[i: Int]: LCRProcess(i, 10)
                 where 0 ≤ i ∧ i < 10;
    C[i: Int]: LCRChannel(i, mod(i+1, 10))
                 where 0 ≤ i ∧ i < 10
```

Figure 2.3: `LCR` system automaton using FIFO channels

The full LCR leader election algorithm is described in Figure 2.3 as a composition of a set of ten process automata connected in a ring by reliable communication channels. The keyword **components** introduces a list of named components: one `LCRProcess` automaton, `P[i]`, and one `LCRChannel` automaton, `C[i]` for each value of `i` as constrained by the **where** predicate. The component `C[i]` is obtained by instantiating the parameters `i` and `j` with the values `i` and `i` + 1 mod 10, so that channel `C[i]` connects process `P[i]` to its right neighbor. The output actions `send(m, i, mod(i+1, 10))` of `P[i]` are identified with the input actions `send(m, i, mod(i+1, 10))` of `C[i]`, and the input actions `receive(m, mod(i+1, 10), i)` of `P[i]` are identified with the output

actions `receive(m, mod(i-1,10), i)` of `C[mod(i-1,10)]`, which is `LCRChannel(mod(i-1,10), 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.

2.8 IOA Toolkit

The IOA language was created as the first step in building an integrated software development environment for distributed systems [45, 46]. The IOA language enables designers to specify automata, their properties, and their relations precisely while spanning many levels of refinement. This environment is intended to support algorithm design, development, testing, and formal verification. The environment, the *IOA toolkit*, connects with both lightweight (syntax checkers, simulators, model checkers) and heavyweight (theorem provers) verification tools. IOA provides a common basis to allow a designer (or design team) to apply any or all of these tools to a single design. Since IOA is a formal language with rigorous, mathematically-based (as opposed to linguistically-based) semantics, the toolkit can automatically translate between the IOA description of an I/O automaton and the equivalent "native" language description for each tool. We hope that the IOA toolkit will encourage distributed system designers to use a variety of formal methods in the design, analysis, and verification of their systems. The toolkit will lower the barriers between disciplines such as model checking and theorem proving and make both types of tools more accessible to distributed system developers. The IOA toolkit is designed to allow developers to explore a design space, refine from high-level conception to low-level implementation, and define and verify system properties — all within a common framework.

The toolkit is divided into a common *front-end* for inputting IOA programs and a variety of *back-end* tools. The front-end consists of the parser, static semantic checker, and the composer. Tools that can be connected at the back-end include an IOA simulator, theorem provers, model checkers, and a compiler.

2.8.1 Checker

The first module to which any IOA program is submitted is the *parser* and *static semantic checker* (or simply the *checker*). In addition to the obvious functions, the checker produces an intermediate representation suitable for use by other tools. This S-expression-based intermediate language (IL) has a simpler parse tree than the more readable IOA source language [102]. As the checker acts as a front-end to just about all other elements of the toolkit, the IL provides a convenient interchange language within the IOA toolkit. A checker prototype has been implemented. The IL representation is semantically equivalent to the source representation. In the future, the checker will be able to emit *proof obligations* that the language specification requires to be true but cannot be easily checked.

These obligations can form the basis of further verification work with other tools.

2.8.2 Composer

The *composition tool (composer)* converts the description of a composite automaton (a collection of other automata) into *primitive form* by explicitly representing its actions, states, transitions, and tasks. The IOA language includes a **components** statement, which defines an automaton to be the composition of referenced automata. The composer expands the composition statement by instantiating and combining the referenced automata as described by the logical composition operation on the model. In the resulting description, the name of a state variable is distinguished by the names of the components from which it arises. The input to the composer must be a *compatible* collection of automata; for example, the component automata should have no common output actions. Note that composition is a semantically "neutral" operation. That is, the I/O automata described by the input program using the **components** statement is equivalent to that described by the output primitive program. Part II of this dissertation describes the design and implementation of the composer tool that has been integrated as part of the checker.

2.8.3 Simulator

The *simulator* [65, 22, 102, 30, 121, 119, 114] performs sample executions of an IOA program, running them on a single machine. The user can help select the executions that are run. IOA programs may describe primitive or composite automata. The simulator checks that proposed invariants are indeed true in all states reached during the course of the simulation. A novel aspect of the simulator is its ability to perform *paired simulations*. In this mode, two automata — a specification and a purported implementation — are run simultaneously. The simulator verifies that a proposed step correspondence maintains the given simulation relation between the two automata (again, only in the states reached during the run).

2.8.4 Theorem Provers

Initially, the IOA toolkit connected to the Larch Prover (LP) [44].[3] Theorem provers can be used to prove validity properties for IOA programs, facts about the datatypes manipulated by programs, invariants of automata, and (forward and backward) simulation relations between automata. In the current implementation, the checker translates IOA descriptions of automata into axioms that can be used by the theorem prover. When a validity condition for an automaton is too hard to establish by static checking, the checker can formulate this condition as a theorem that must be proved. It can also formulate sets of lemmas that imply that asserted invariants and simulation relations are

[3]Independent of this toolkit project, Devillers has used the IOA language specification [43] to build a tool that translates IOA into input for PVS [33].

indeed invariants and simulation relations. Users can interact with a theorem prover to prove that the lemmas follow as consequences of the axioms. In some cases, the theorem prover can prove a lemma automatically, but usually the user must interact with the prover to suggest proof strategies and other useful information.

2.8.5 Model checkers

Model checkers provide an approach to validation that is complementary to theorem proving and to simulation. They work completely automatically, and can be used to validate all executions of a finite-state system with a sufficiently small number of states. The IOA toolkit is designed to utilize existing model checkers. The first model checking prototype interface was targeted to Spin [60]. A model checker can be used as a validator of invariants and possible simulation relations provided by the user.

Uppaal is a tool that assists in the development of timed systems, giving users simulation and verification capacities to guarantee timing properties in computer programs [76]. TIOA is under development as extension to IOA for describing timed automata [66, 67]. The TIOA-to-UPPAAL translator allows the simulation of IOA and TIOA programs in UPPAAL's easy-to-use interface, and the checking of their properties with UPPAAL's model checker [105].

2.8.6 Invariant discovery tools

Daikon is a tool that examines executions of programs in order to suggest possible invariants of the program [36]. The IOA simulator has been instrumented to produce traces suitable for Daikon to examine [30]. Ne Win *et al.* investigated the use of the automatically discovered invariants to help increase the automation of theorem proving [121, 122].

2.8.7 IOA Compiler

All of the above tools work entirely within formal frameworks. These tools allow designers to perform a variety of design, specification, and verification tasks. Even with all this technology and these tools, however, the task of the distributed system builder remains outside the model. The builder must translate the designers' requirements (now formally described) into a standard imperative programming language. In essence, the system builder must start over and recode the whole project. As a result, a disconnect exists between the specification and the actual running code. Part of the goal in designing the IOA language and toolkit is to bridge this gap. The main contribution of this dissertation is a tool that bridges that gap, an *IOA compiler*.

The IOA compiler translates a restricted subset of IOA programs into Java. The resulting code runs on a collection of workstations communicating using standard networking protocols. The

remainder of this dissertation describes the design and implementation of the IOA compiler and argues that the compiler bridges the gap between specifications with formal proofs of correctness and running code by preserving the safety properties of IOA specifications in the generated Java code.

Chapter 3

Structuring the Design

Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.

— Alan J. Perlis [99]

IOA can describe systems architected in just about any configuration a system designer can dream up, including completely centralized designs, shared memory implementations, or message passing arrangements. However, the IOA compiler targets only message passing systems. The goal is to create a running system consisting of the compiled code and the existing MPI service that faithfully emulates the original distributed algorithm written in IOA. According to the semantics of IOA, the individual actions of the algorithm, everywhere in the system, are atomic and execute sequentially. In the running system, each IOA atomic action is expanded into a series of smaller steps corresponding to Java operations. The steps corresponding to different atomic actions may execute in an interleaved fashion, or concurrently. The IOA compiler must ensure that the effect as seen by external users of the algorithm is "as if" the high-level actions happened atomically.

One approach to preserving the externally visible behavior of the system is to ensure atomicity by synchronizing among processes running on different machines, thus reducing the possible sources of concurrency. This approach to implementing I/O automata (by hand coding) was taken, for example, by Cheiner and Shvartsman [23, 25, 24]. Such an approach has the advantage that it is very general. The implemented automata do not need to be structured in the same way as the hardware on which they run. However, such global synchronization is expensive. Before an automaton at one node can execute an external action, it must coordinate with the automata at one or more other nodes. This coordination requires extra messages and blocking the execution of the

automaton until synchronization is complete.

A major challenge in our work is to achieve the appearance of globally-atomic IOA steps *without any synchronization between processes running on different machines.* Rather than attempt a generalized approach, we require the programmer to match the system design to the target language, hardware, and system services before attempting to compile. The initial target environment for the IOA compiler is a group of networked workstations.

Each host runs a Java interpreter with a console interface and communicates with other hosts via (a subset of) the Message Passing Interface (MPI) [41, 4]. (By "console" we mean any local source of input to the automaton. In particular, we call any input that Java treats as a data stream — other than the MPI connection — the console.) We are able to preserve the externally visible behavior of the system without synchronization overhead because we require the programmer to explicitly model the various sources of concurrency in the system: the multiple machines in the system, the communication channels, and the console interface to the environment.

These requirements can be divided into two categories: restrictions on the form of the IOA system description and requirements on the semantics of the system — that is, syntactic restrictions and proof obligations. The syntactic restrictions define which elements and structures of the IOA language may be used to describe a system admissible for compilation. Admissible programs use only imperative IOA constructs and are composed in "node-channel form" using mediator and interface automata. We discuss these requirements below in Sections 3.1, 3.2, and 3.5. In addition, admissible programs must be "scheduled" as discussed in Chapter 5.

Proof obligations are criteria an IOA program must meet if the system is to behave as the programmer expects. A proof of correctness for the system design should include these additional constraints in the specification of correct behavior of the system. In particular, an IOA system submitted for compilation must be input-delay insensitive. That is, the program must behave correctly (as defined by the programmer) even if its inputs from the local console are delayed. This is a technical constraint that most interesting distributed algorithms can be altered to meet. We discuss the console interface to IOA programs in Sections 3.3 and 3.4.

3.1 Imperative IOA programs

As mentioned in Section 2.6, IOA supports both operational and axiomatic descriptions of programming constructs. The prototype IOA compiler translates only imperative IOA constructs. Therefore, IOA programs submitted for compilation cannot include certain IOA language constructs. The automaton state declaration cannot include **initially** clauses which can assert arbitrary predicates on the initial values of state variables. Effects clauses cannot include **ensuring** clauses that relate pre-states to post-states declaratively. Throughout the program, predicates must be quantifier free.

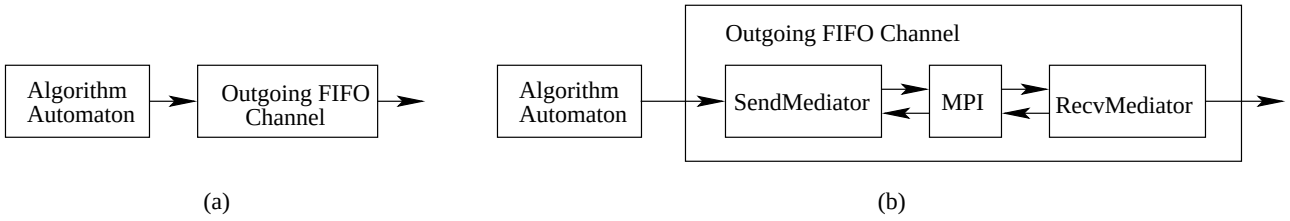

Figure 3.1: Auxiliary automata mediate between MPI and node automata. (a) Reliable, FIFO channel defines the desired behavior. (b) The composition of MPI and the mediator automata implements the reliable, FIFO channel.

Currently, the compiler handles only restricted forms of loops that explicitly specify the set of values over which to iterate.

Later versions of the compiler may support annotations of the IOA program to provide witnesses for certain classes of existentially quantified predicates and iterators for certain finite types of loop or universally quantified variables. (These annotations would be an extension to the NDR language discussed in Chapter 5.)

3.2 Node-Channel Form

We require that systems submitted to the IOA compiler be described in *node-channel* form. Specifically, the source IOA specification consists of a collection of N *algorithm automata* connected by up to N^2 channels. Each algorithm automaton describes the computation performed at one node in the system design. As in the LCR example, differently parameterized instances of the same algorithm automaton may be run at different nodes.

3.2.1 Abstract Channels

While code generated by the IOA compiler must interface with MPI, the intricacies of interfacing with the MPI system are somewhat distracting to the distributed system designer. So, for convenience, we specify a simpler *abstract channel* interface that allows programmers to design their systems assuming the existence of reliable, one-way FIFO channels similar to those specified in Figure 3.1. In this simple interface, the `RECEIVE` input action connects with incoming channels and the `SEND` output actions connects with outgoing channels. We divide the external actions of an algorithm automaton into two categories. The `SEND` and `RECEIVE` actions are the *network actions*. *Console actions* are all the other external actions.

The actual compiled code must still interface with MPI. Therefore, we define auxiliary IOA automata to mediate between MPI and the network actions of the algorithm automaton so as to implement a FIFO channel. The `ReceiveMediator` automaton mediates between the algorithm automaton and an incoming channel, while `SendMediator` handles messages to outgoing channels.

Each of the N node programs is connected to up to $2N$ mediator automata (one for each of its channels). Figure 3.1 depicts the relationship of the algorithm automaton and an abstract outgoing channel automaton and how mediator automata are composed with MPI to create that channel. Chapter 4 details the precise abstract channel interface available to IOA programs and shows how combinations of mediator automata and MPI provide that interface by connecting to the network actions. The console actions are discussed below.

3.3 Handshake Protocols

The I/O automaton model requires that input actions are always enabled. However, Java programs are not input enabled. Usually, a Java program receives input only when the program asks for it by invoking a method. We model this behavior in IOA by a simple handshake protocol. An automaton signals its readiness to receive input from its environment by executing an output action. Thereafter, the environment may respond by a single invocation of an input action of the automaton. (That invocation is an output action from the viewpoint of the environment. For clarity, we stick to the viewpoint of the automaton.) The environment may not invoke a second input until it receives a second output signal. This pairing of output and input actions models the call and return of a Java method.

To be faithful to the Java execution model, after the output signal (method call) all other activity in the automaton must stop until the expected input (method return) occurs. We could model this stoppage by disabling all locally controlled actions of the automaton. As we show in Chapter 7, it suffices for our purposes simply not to schedule any actions in this interval. In addition, every input action to the automaton must be controlled by a similar handshake protocol.

Given any two interacting automata, it is simple to augment the interface between them with a formal handshake protocol. Let `A` be an automaton and `Env` be its environment automaton. We want to augment the two automata to require that `Env` generate inputs to `A` only after `A` signals it is ready to receive them.

We augment the state of the environment automaton `Env` with an additional boolean variable `enabled` and an input action `inReady`. Initially, `enabled` is false. The `inReady` input action sets `enabled` to true. We strengthen the precondition of each output action to require `enabled` to be true and extend its effect to assign false to `enabled`. Thus, in any valid execution of `Env`, any two output actions must be separated by an `inReady` input action. These augmentations are shown in Figure 3.2.

Similarly, we augment the state of the node automaton `A` that connects to the environment automaton with an additional boolean variable `signalled` and an output action `inReady`. Initially, `signalled` is false. The precondition of the `inReady` output requires `signalled` to be false while the

```
automaton Env
  signature
     output pi
     input inReady
     ...
  states
     enabled: Bool := false,
     ...
  transitions
    input inReady
      eff enabled := true
    output pi
      pre enabled;
          ...
      eff enabled := false;
          ...
```

Figure 3.2: Example augmentations to the environment automaton to perform a handshake protocol.

effect merely toggles `signalled` to true. We extend the effects of each input action to set `signalled` to false. If some other local processing must be performed before the node automaton is ready for input, that assignment can instead appear in some other transition in the automaton. However, that additional processing must be purely local. No output can be generated before the flag is toggled. Thus, in any valid execution of `A`, any two instances of the `inReady` output action must be separated by an input action. These augmentations are outlined in Figure 3.3.

```
automaton A
  signature
     input pi
     output inReady
     ...
  states
     signalled: Bool := false,
     ...
  transitions
    output inReady
      pre ¬signalled
      eff signalled := true
    input pi
      eff signalled := false;
          ...
```

Figure 3.3: Example augmentations to a node automaton to perform a handshake protocol.

3.4 Console Interface

We want to provide the IOA programmer with the convenience of the simplest possible interface to the console but we do not want to force the programmer to model the intricacies of the Java I/O interface. Instead, we allow the programmer to specify an arbitrary interface to the console for

that node. That is, the console actions of a program may include any input or output actions with (almost) any parameters.[1]

As we describe in Section 6.4.2, the IOA compiler uses S-expressions to describe arbitrary IOA actions and datatypes externally. One approach to modeling the console interface would be to specify a series of detailed, interacting automata that read and parse individual characters, whole lines, well-formed S-expressions, etc. Instead, we choose to model the console at a higher level of abstraction. We consider the reading and parsing of an IOA action and its parameter values to happen in a single step. We fold all those low-level steps into a single handshake between the console and the automaton.

As mentioned, simple handshake protocols can bring the execution of an automaton to a halt while the automaton waits for input. Similarly, simple approaches to I/O in Java programs can bring the program to a halt while it blocks on a method call. Two standard techniques for avoiding such stoppages are the use of nonblocking calls and multithreading. Nonblocking calls limit the time the program stops by returning "quickly". The data returned by such calls might not be substantive input to the program. Often, the returned value just indicates that no substantive data was ready at the time. Repeated invocations of nonblocking calls (called polling) can be interleaved with other work. Multithreading permits part of the program to continue executing concurrently while another part blocks until substantive data is available.

In the IOA compiler, we use both these techniques. We use nonblocking MPI calls to implement the abstract channel interface as described in Chapter 4. We use both multithreading and polling to implement the console interface.

The program emitted by the IOA compiler has three threads. The main thread performs the work of the program as submitted to the compiler. An input thread waits for input from the console, parses it, and then copies the invocation into an internal buffer. The main thread must poll this internal buffer to see if input has arrived. An output thread waits for output from the program, generates an S-expression representation, and then sends that to the console.

3.4.1 Buffer Automata

One consequence of this design is that console inputs might not be handled immediately upon arrival. Therefore, we require that the IOA system submitted for compilation be designed so that its safety properties hold even when console inputs to any node in the system are delayed. Specifically, the programmer must write IOA programs so that the algorithm is correct even when each node automaton is composed with the particular kind of buffer automaton we introduce in this section.

[1] There is a technical constraint that a console action include at least the node identifier among its parameters. That parameter prevents like-named input actions from collapsing into a single action when considering the composition of the entire system as we do in Section 7.1. In like-named output actions, the identifier prevents compositional incompatibility across node automata.

Informally, we say such a system is input-delay insensitive. By "correct", we mean that the entire system of nodes and channels must exhibit only behaviors the programmer wishes to allow. The strongest notion of input-delay insensitivity would be that the system exhibits no new externally visible behaviors when each node automaton is composed with its buffer automaton.

Even though the Java implementation is generic, we specify the correctness condition for each node automaton with a buffer automaton specific to the console interface of that algorithm automaton. The buffer automaton for the `LCRProcess` algorithm automaton is shown in Figure 3.4.

The signature of each buffer automaton mimics the console actions of its corresponding algorithm automaton. That is, for each console action `pi` of the algorithm automaton, the buffer automaton has a corresponding action `pi` of the same kind with the same parameters. In addition, the buffer automaton implements the handshake protocol for inputs from the environment described above. The buffer automaton has state variable `signalled` and a signaling output action `inReady`. This action models the single Java method call used to initiate all console input to the automaton. The input is differentiated into individual IOA action invocations by parsing the S-expression received.

The buffer automaton implements the handshake protocol for outputs to the environment too. In this case the roles of "node" and "environment" are reversed. Thus, the buffer automaton also has a state variable `enabled` and a signaling input action `outReady` models the return from the sole Java method call used to initiate all console output from the automaton. (The name is changed from `inReady` to avoid conflict.) Since some local processing must occur before the automaton is ready to accept new input, falsifies the `signalled` flag in the internal `appendInvocation` action rather in an input action.

In the case of the `LCRProcess` algorithm automaton, there are two console actions: the input action `vote` and the output action `leader`. Therefore, the `LCRProcessInterface` buffer automaton has inputs `vote` and `outReady` and outputs `leader` and `inReady`.

Each buffer automaton defines the specific IOA sorts used to represent the input and output invocations. Each invocation is represented as a tuple of an action label and a sequence of parameters. That tuple sort is called `IOA_Invocation`. Action labels in such tuples are collected into an enumeration sort named `IOA_Action`. Parameters are represented by a sort `IOA_Parameter` that is the union of all sorts that may appear as a parameter of any console interface action of the algorithm automaton.

The `LCRProcessInterface` buffer automaton defines `IOA_Action` to be the enumeration of `vote` and `leader`. Since both actions have only one parameter and that parameter is an integer in both cases, the union `IOA_Parameter` has only one possible tag: `Int`.

The effect of the buffer input action is to construct an `IOA_Invocation` that represents the algorithm input action and to toggle the `signalled` and `valid` flags. The latter flag is true when a new invocation has arrived but has not yet been inserted into the buffer. This action abstracts away the

```
type IOA_Invocation = tuple of action: IOA_Action, params: Seq[IOA_Parameter]
type IOA_Parameter = union of Int: Int
type IOA_Action = enumeration of leader, vote
automaton LCRProcessInterface(i: Int, ringSize: Int, name: Int)
  signature
    input
      vote(I0: Int) where I0 = i,
      leader(I5: Int) where I5 = i,
      outReady(I0: Int) where I0 = i
    output
      leader(I5: Int) where I5 = i,
      vote(I0: Int) where I0 = i,
      inReady(I0: Int) where I0 = i
    internal
      appendInvocation(I0: Int) where I0 = i
  states
    valid: Bool := false,
    signalled := false,
    enabled := false,
    invocation: IOA_Invocation,
    stdin: LSeqIn[IOA_Invocation]:= {},
    stdout: LSeqOut[IOA_Invocation]:= {}
  transitions
    output inReady
      pre ¬signalled
      eff signalled := true
    input vote(I0)
      eff invocation := [vote, {} ⊢ Int(I0)];
          valid := true;
    internal appendInvocation(I0)
      pre valid
      eff stdin := stdin ⊢ invocation;
          valid := false;
          signalled := false;
    output vote(I0)
      pre stdin ≠ {} ∧
          ((((head(stdin).action) = vote) ∧
          (len(head(stdin).params)) = 1) ∧
          (tag(head(stdin).params[0])) = Int) ∧
          (head(stdin).params[0].Int) = I0
      eff stdin := tail(stdin)
    input leader(I5)
      eff stdout := stdout ⊢ [leader, ({}) ⊢ Int(I5)]
    output leader(I5)
      pre enabled ∧
          stdout ≠ {} ∧
          (((head(stdout).action) = leader) ∧
          (len(head(stdout).params)) = 1) ∧
          (tag(head(stdout).params[0])) = Int) ∧
          (head(stdout).params[0].Int) = I0
      eff stdout := tail(stdout);
          enabled := false
    input outReady
      eff enabled := true
```

Figure 3.4: Complete IOA specification the `LCRProcessInterface` buffer automaton

Java I/O interface and S-expression parsing.

Once an input invocation has been received (*i.e.*, `valid` is true), the internal `appendInvocation` action may append the invocation to the `stdin` buffer and reset `valid` to false. As we show in Section 7.2, this append step must be in a step separate from the input action in order to model locking `stdin` correctly.

The effect of the buffer output action is to dequeue an `IOA_Invocation` from the `stdout` buffer. Note that all invocations come in through the `stdin` queue and go out through the `stdout` queue. Therefore an output action `pi` is only enabled when an `IOA_Invocation` for a `pi` action is at the head of the `stdout` queue. For example, the output action `leader` is only enabled when `stdout` is not empty, the `IOA_Invocation` at the head of the queue has an `action` field specifying `leader` and a `params` field of one element, whose tag is `Int` and whose value is the value to be output.

Finally, for every console action `pi` of the algorithm automaton, the buffer automaton has a second action `pi` with the kind (input or output) inverted. These actions permit the buffer automaton to inserted between the algorithm automaton and any environment with which it might be composed, as illustrated in Figure 3.6. These inverted actions dequeue invocations from the `stdin` buffer or enqueue invocations onto the `stdout` buffer. The preconditions for the output action that dequeues invocations from `stdin` mirror those for the output action that dequeues invocations from `stdout`. For example, the output action `vote` is only enabled when `stdin` is not empty, the `IOA_Invocation` at the head of the queue has an `action` field specifying `vote` and a `params` field of one element, whose tag is `Int` and whose value is the value to be output.

Technically, such a buffer automaton is not a valid IOA program because it has overlapping, like-named input and output actions. (See Section 11.4.) One way to get around this technical constraint would be to rename the console actions of the algorithm automaton and the corresponding inverted actions of the buffer automaton to some fresh name. We ignore this issue because, as explained below, the actual external interface of the buffer automaton is only conceptual.

Essentially, the buffer automaton has two flows through it. Every input action constructs a representation of itself. That representation is stored in a buffer (either `stdin` or `stdout`) for some time. When the invocation is at the head of its queue it is output by the appropriate output action. These two flows are almost entirely symmetrical (ignoring the extra stutter step of the append action). The real difference between them is external to the buffer automaton itself. The difference arises from which end of each flow is connected to the algorithm automaton and which end is connected to the console.

3.4.2 Interface Generator

As mentioned above, the compiler implements a multithreaded translation of IOA programs. As we describe in Section 6.4.2, the input thread and output threads are treated as special cases. In fact,

the implementation of these I/O threads is independent of the algorithm being compiled. The above description of buffer automata is simply our model of (or correctness condition for) the behavior of the I/O threads.

Actually, buffer automata as described above include *more* than the behavior of the I/O threads. That is, not all the actions we describe are implemented by the I/O threads. In particular, the "inverted' actions are implemented by the main thread. Notice this means that access to the `stdin` and `stdout` queues is shared across threads. Thus, items enqueued onto `stdin` by the input thread are dequeued from it by the main thread. Similarly, items enqueued onto `stdout` by the main thread are dequeued by the output thread. Only those flipped actions are needed for compilation.

```
type Status = enumeration of idle, voting, elected, announced
type IOA_Invocation = tuple of
action: IOA_Action, params: Seq[IOA_Parameter]
type IOA_Parameter = union of Int: Int
type IOA_Action = enumeration of RECEIVE, SEND, leader, vote
automaton LCRProcessInterface(i: Int, ringSize: Int, name: Int)
  signature
    input
     leader(I5: Int) where I5 = i
    output
     vote(I0: Int) where I0 = i
  states
    stdin: LSeqIn[IOA_Invocation]:= {},
    stdout: LSeqOut[IOA_Invocation]:= {}
  transitions
    output
      vote(I0)
        pre stdin ≠ {} ∧
            ((((head(stdin).action) = vote) ∧ (len(head(stdin).params)) =
            1) ∧ (tag(head(stdin).params[0])) = Int) ∧
            (head(stdin).params[0].Int) = I0
        eff stdin := tail(stdin)
    input
      leader(I5)
        eff stdout := stdout ⊢ [leader, ({}) ⊢ Int(I5)]
```

Figure 3.5: IOA specification for the abbreviated `LCRProcessInterface` buffer automaton as produced by the interface generator

The *interface generator* tool implemented by Michael Tsai can be used to automatically produce from an algorithm automaton the part of its buffer automaton that is needed for compilation [119]. While the complete buffer automaton `LCRProcessInterface` is shown in Figure 3.5, the automaton actually produced by the interface generator is shown in Figure 3.5. The role of the special sorts `LSeqIn` and `LSeqOut` in code generation is also explained in Section 6.4.2. Semantically, they are identical to the standard IOA sort `Seq`. Note, the transitions with duplicate names are omitted from this version of the buffer automaton, avoiding the technical problem mentioned above. The `Status` type definition appearing in the figure is copied by the interface generator from the `LCRProcess` automaton. It is not needed. Similarly, the inclusion of `SEND` and `RECEIVE` in the `IOA_Action` enumeration is

an artifact of the current implementation.

3.5 Composition

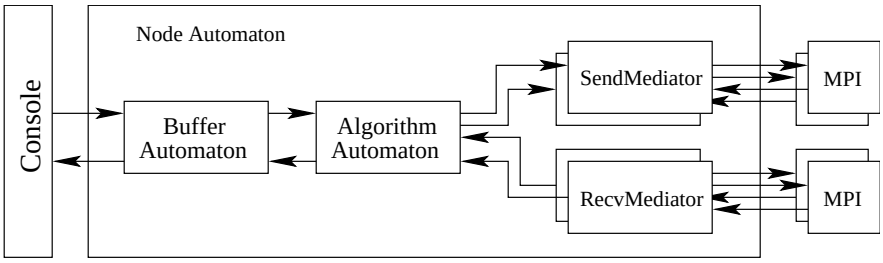

Figure 3.6: An annotated node automaton composed of a buffer automaton, an algorithm automaton, and mediator automata is the program input to the IOA compiler.

The completed design is called the *composite node automaton* and is described as the composition of the algorithm automaton provided by the programmer with the mediator automata introduced above in Section 3.2.1 and the buffer automata produced by the interface generator as described in Section 3.4.2. The full definition of the mediator automata is given in Section 4.5. A *composer* tool [118] expands this composition into a new, equivalent IOA program in primitive form (*i.e.*, without any **components** statements). The design of the composer tool is the subject of Part II of this dissertation.

The resulting *node automaton* describes all computation local to one machine. The node automaton communicates with other nodes only via the channel automata. The node automaton (annotated as described in Chapter 5) is the actual input program to the IOA compiler. The compiler translates each node automaton into its own Java program suitable to run on the target host. Figure 3.6 depicts the composition that produces a node automaton.

In the node automaton, both the console and network interfaces of the algorithm automaton are entirely hidden. The console actions are matched with actions from the buffer automaton. The network actions are matched with actions from the mediator automata. As a result, the entire external interface of the algorithm automaton as designed by the programmer has been encapsulated inside standard actions provided by the compiler or associated libraries and tools. This fact allows us to handle these known quantities as special cases during compilation while providing precise formal semantics for verifying algorithm automata.

For the LCR example, the composite node automaton is shown in Figure 3.7. In that figure, the composite node `LCRNode` is the result of composing one instance of the `LCRProcess` algorithm automaton with one instance of the `LCRProcessInterface` buffer automaton and `MPIsize` instances each of the `ReceiveMediator` and `SendMediator` interface automata. The resulting composite node program is parameterized by the node identifier `MPIrank`, the number of nodes in the ring `MPIsize`,

```
axioms Infinite(Handle)
automaton LCRNode(MPIrank, MPIsize, name: Int)
  components
    P: LCRProcess(MPIrank, MPIsize, name);
    RM[j: Int]: ReceiveMediator(Int, Int, j, MPIrank)
                 where j = mod(MPIrank-1, MPIsize);
    SM[j: Int]: SendMediator(Int, Int, MPIrank, j)
                 where j = mod(MPIrank+1, MPIsize);
    I: LCRProcessInterface(MPIrank, MPIsize, name)
    hidden SEND(m, i, j), RECEIVE(m, j, i), vote(i), leader(i)
```

Figure 3.7: IOA specification for one node of an `LCR` system

and the name of the node. The primitive form of the `LCRNode` automaton output by the composer tool (the node automaton for the LCR system) is shown in Appendix A.1.

Chapter 4

Implementing Abstract Channels with MPI

It is easier to change the specification to fit the program than vice-versa.

— Alan J. Perlis [99]

In an ideal world, designing distributed systems that behave both correctly and efficiently would be easy. In reality, one of these goals is often achieved at the expense of the other. For example, a correct design is often most easily achieved by limiting the concurrency in and, hence, the efficiency of a system. As we described in Chapter 3, our approach to balancing these goals is to give programmers the appearance of globally-atomic IOA steps while not using any global synchronization. The cost of this approach is that we require the programmer to match the system design to the target computing environment. For the communication service portion of a system design, our target environment is the Message Passing Interface (MPI). Thus, the code generated by the IOA compiler must interface with MPI.

In other words, the running system must link to MPI libraries. The behavior of these libraries is described in the message-passing interface standard [41]. The Java interface we use to access the MPI libraries is described in Baker *et al.* [4]. We summarize the relevant features of MPI in Section 4.1. In Section 4.2 we make all our assumptions about the behavior of these libraries explicit by modeling MPI as an IOA program `MPI`. In Section 4.3 we make explicit the requirements for an clients of the MPI libraries by defining the `SendClient` and `ReceiveClient` automata. Any program connecting to the MPI libraries (*e.g.*, a node automaton) must adhere to one (or both) of these specifications.

Although our approach requires the system designer to structure a system in node-channel form, further requiring the programmer to write code specifically to interact with the MPI libraries is more restrictive than necessary. Such a requirement overshoots the balance we aim to achieve. It would require programmers to complicate system designs unnecessarily and, thus, it would make achieving correct systems harder. Instead, we allow programmers to target a simpler channel interface and semantics. We precisely define these simpler *abstract channels* by specifying the `AbstractChannel` automaton in Section 4.4.

We are able to achieve the semantics of these abstract channels while connecting to MPI by interposing auxiliary automata to mediate between MPI and the algorithm automaton. The mediator automata must fulfill three goals. First, they must interact correctly with the `MPI` automaton. That is, the network external actions must correspond to the appropriate external actions of the `SendClient` or `ReceiveClient` automata. Second, the mediator automata must provide the programmer the desired abstract channel interface. That is, the rest of the external actions must correspond to the appropriate external actions of the `AbstractChannel` automaton. Third, the mediator automata must do the right things. That is, the appropriate combination of the mediator automata and `MPI` must act like the `AbstractChannel` automaton. We define the two mediator automata used in our design and show they have the appropriate interfaces in Section 4.5.

We make the notion of "appropriate combination" of automata precise in Section 4.6 by defining a composite automaton `CompositeChannel` with components based on `MPI`, `SendMediator`, and `ReceiveMediator`. In Section 4.7 we prove `CompositeChannel` "does the right thing" by demonstrating a simulation relation from `CompositeChannel` to `AbstractChannel`, thus showing that the set of traces of the former is a subset of the set of traces of the latter. Section 4.8 suggests how one might connect the IOA compiler to network services other than MPI.

4.1 MPI

The Message Passing Interface (MPI) is a communication service designed for programming machine clusters and parallel computers. MPI provides many communication modes and capabilities. Communication can be blocking or nonblocking. Messages can be sent point-to-point, be broadcast, or be multicast. For the simple point-to-point abstract channels in our design, it suffices to use only a few of the many available MPI calls.

To avoid any global synchronization, we use communication primitives that require only local computation and assume asynchronous channels. In particular, `MPI` models the behavior of the MPI system in response to four basic calls used to send and receive messages: **Isend**, **test**, **receive**, and **Iprobe**. We use a handful of additional calls during system initialization and tear-down. However, since the IOA model of computation is static and does not include these phases, these additional

calls are not included in any automata or proofs.

We use the MPI asynchronous send communication mode (and associated **Isend** primitive) because such calls are nonblocking. That is, the calls do not depend on any action by another MPI client. In other words, nonblocking calls to MPI are guaranteed to return in finite time even if no other MPI client makes progress.

Although nonblocking communication has desirable properties, the nonblocking communication interface is more complicated than that for blocking communication. In particular, the sending process must coordinate with the MPI communication system to manage the shared memory space used to buffer outgoing messages. Since communication may take an arbitrary amount of time but nonblocking calls must return immediately, an arbitrarily large amount of memory may be necessary to buffer pending messages. MPI requires the sending process to provide the buffer for each message. (This sensible policy pushes the resource requirements back on the process generating the messages.) MPI uses the buffer for as long as necessary before releasing it back to the sending process. Coordinating the allocation and release of message buffers to and from MPI requires using two calls: **Isend** to allocate the buffer (and send the message contained in it) and **test** to poll for the release of the buffer from MPI back to the sending process.

MPI uses *requests* to identify communication operations (*e.g.*, sending a message) and match the shared resources (*e.g.*, message buffer) used with the operations across multiple calls. Memory managed (only) by MPI is not directly accessible to the user, and objects stored there are said to be *opaque*. Opaque objects are referenced via *handles* (*i.e.*, pointers) passed between MPI and applications. Handles may be compared and copied (assigned) but not otherwise manipulated. MPI provides a "null handle" constant for each object type. Comparisons to this constant are used to test for the validity of the handle.

4.1.1 Method descriptions

Four basic methods are sufficient to constitute a simple, nonblocking, send-receive interface to the MPI communication system: **Isend**, **test**, **receive**, and **Iprobe**.[1] The interaction of clients and the MPI system can be described informally as follows.

Isend initiates the delivery of a message from one MPI process to another. It allocates a communication request within the MPI communication system and associates it with a handle. The request (as named by its handle) can later be used to **test** for completion of the **Isend**. Invoking **Isend** indicates that the system may start copying data out of the message buffer. The sending process is required not to access any part of the buffer after an **Isend** operation is called until the **Isend** is *complete*. Completion is an event internal to the communication subsystem. The occurrence of this

[1] The prefix of I (for immediate) indicates that a method is nonblocking. Don't ask me why it's **test** and not **Itest**.

event can be detected with inquiries (*e.g.*, **test**). When an **Isend** operation is complete the sender is free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). Completion does not indicate that the message has been received; rather, the message may have been buffered by the communication subsystem.

receive retrieves a message from the communication system. **receive** is blocking: it returns only after the message buffer contains the newly received message. The return of a **receive** operation indicates the receiver is now free to access the received message. It does not indicate that the matching **Isend** operation has completed (but indicates, of course, that the **Isend** was initiated).

test returns true if the operation associated with its `Request` argument is complete.[2] In such a case, the associated request object is deallocated by the method. In addition, the client may now reuse or deallocate the associated message buffer. Otherwise, **test** returns false. An invocation of **test** with a null or inactive request argument is allowed. An inactive handle is a handle that does not name an existing request object. In such a case, the operation returns true.

Iprobe returns with its `flag` argument true if there is a message ready to be received.

We use several additional MPI methods during initialization and finalization. These calls are each used only once or twice and do not appear in our steady state model of the communication service. The methods are **init**, **finalize**, **Barrier**, **Rank**, **Size**, **Get_processor_name**. Each node must invoke the **init** method before invoking any other MPI method. Once the **finalize** method has been invoked, no other MPI method may be invoked. All pending communication must be complete for **finalize** is invoked. The **Barrier** method does not return until every node in the system has invoked a **Barrier** call. The **Rank** method returns an integer distinct from the rank returned at any other node. The **Size** method returns the total number of nodes in the system. The **Get_processor_name** returns a string naming the host on which the node is running.

4.1.2 Resource limitations

The MPI specification states that every pending communication operation consumes finite resources and thus may fail. Unfortunately, this blanket statement does not help to identify what resources are consumed or how much is available. Frankly, the specification could not say much more because these costs are obviously implementation dependent.

In some situations a lack of buffer space leads to deadlock situations. In fact, we have verified that in our testbed **Isend** will block after about 512 operations if no matching **receive** operations

[2]Note, the `Request` argument is actually a handle that names a request object internal to MPI.

are initiated.[3] Some types of resource dependency may be deduced. For example, while **Isend** avoids requiring the communication service to allocate buffer space for message storage, clearly the communication system *must* expend some amount of resources storing the characteristics of each message. This cost is likely to be constant per uncompleted **Isend**. Currently our model allows an infinite number of uncompleted **Isend**s. Clearly any real implementation will limit this number.

4.2 MPI Specification Automaton

Although MPI is implemented as a monolithic communication substrate, we model MPI as a collection of n^2 point-to-point channels connecting n client computation nodes (including self-loops). The `MPI` automaton shown in Figures 4.1 and 4.2 specifies one asynchronous one-way channel between two client nodes. The sending node uses the **Isend** and **test** actions to invoke MPI; the receiving node uses **Iprobe** and **receive**. The automaton specifies the behavior of MPI upon receiving these inputs. Section 4.3 details our assumptions about client behavior.

The `MPI` automaton explicitly details our assumptions about the behavior or the system in response to the four MPI methods our design invokes. For example, `MPI` outputs a `resp_`* action only in response to the corresponding input action. That is, methods do not return unless they have been invoked. Furthermore, **Isend**, **test**, and **Iprobe** do not block. Thus, `resp_Isend`, `resp_test`, and `resp_Isend` actions are guaranteed to become enabled in a finite number of steps after their corresponding call input actions even if no other inputs to `MPI` occur in the execution.

Sorts Our MPI specification automaton uses several additions to the built-in IOA sorts. First, we introduce the `Handle` sort to model MPI handles (see Figure 4.3). The `Handle` sort is an infinites collection of unique items. The only operations allowed on handle are handle generation and (implicitly defined) equality checking. Second, our specification defines several shorthand sorts (tuples and enumerations) for tracking the current state of the system. Sorts `rCall` and `sCall` enumerate the possible method invocations by sending and receiving process, respectively. A `receiveStatus` or `sendStatus` tuple encapsulates the status of the MPI system handling one of those method invocations. The `Request` tuple associates message content with a unique handle for naming and status information about completion and delivery. These latter two sorts are defined in an LSL trait parameterized by the `Msg` sort (see Figure 4.4).

The `MPI` automaton is parameterized by the sort `Msg` specifying the messages it transports, the sort `Node` specifying the nodes it connects, and the identifiers for the nodes at its endpoints (the sender `i` and the receiver `j`).

[3] Chapter 8 describes our experimental testbed.

```
axioms Infinite(Handle)
axioms sStatus for sStatus[__]
axioms sStatus for Request[__]

type rCall = enumeration of idle, receive, Iprobe
type sCall = enumeration of idle, Isend, test
type rStatus = tuple of call: rCall

automaton MPI(Msg, Node:Type, i, j:Node)

  signature
    input Isend(m: Msg, const i, const j)
    output resp_Isend(handle:Handle, const i, const j)
    input test(handle:Handle, const i, const j)
    output resp_test(flag:Bool, const i, const j)

    internal complete(request:Request[Msg], const i, const j)
    internal move(request:Request[Msg], const i, const j)

    input receive(const i, const j)
    output resp_receive(m: Msg, const i, const j)
    input Iprobe(const i, const j)
    output resp_Iprobe(flag:Bool, const i, const j)

  states
    channel: Set[Request[Msg]] := {},
    handles: Seq[Handle] := {},
    toRecv: Seq[Msg] := {},
    sendStatus: sStatus[Msg] := choose,
    receiveStatus: rStatus := [idle]
    initially sendStatus.call = idle

  transitions
    input Isend(m, i, j)
      eff sendStatus.call := Isend;
          sendStatus.msg := m

    output resp_Isend(handle, i, j)
      pre sendStatus.call = Isend;
          ∀ r:Request[Msg] (r ∈ channel ⇒ r.handle ≠ handle)
      eff sendStatus.call := idle;
          handles := handles ⊢ handle;
          channel := insert([sendStatus.msg,handle,false,false], channel)

  input test(handle, i, j)
      eff sendStatus.call := test;
          sendStatus.handle := handle

  output resp_test(flag, i, j; local request:Request[Msg])
      pre sendStatus.call = test;
          request.handle = sendStatus.handle;
          request ∈ channel ⇒ flag = request.complete;
          ¬(request ∈ channel) ⇒ flag = true
      eff sendStatus.call := idle;
          if request.complete then channel := delete(request, channel) fi
```

Figure 4.1: Signature, states, and sender-side transitions of MPI System Automaton, MPI

```
internal move(request, i, j)
  pre request ∈ channel;
      request.sent = false;
      request.handle = head(handles)
  eff toRecv := toRecv ⊢ request.msg;
      handles := tail(handles);
      channel :=
             insert([request.msg, request.handle, false, true],
                    delete(request, channel))

internal complete(request, i, j)
  pre request ∈ channel;
      request.sent = true;
      request.complete = false
  eff channel :=
              insert([request.msg, request.handle, true, true],
                     delete(request, channel))

input receive(i,j)
  eff receiveStatus.call := receive

output resp_receive(m, i, j)
  pre receiveStatus.call = receive;
      head(toRecv) = m
  eff toRecv := tail(toRecv);
      receiveStatus.call := idle

input Iprobe(i, j)
  eff receiveStatus.call := Iprobe

output resp_Iprobe(flag, i, j)
  pre receiveStatus.call = Iprobe;
      toRecv ≠ {} ⇒ flag = true;
      toRecv  = {} ⇒ flag = false;
  eff receiveStatus.call := idle
```

Figure 4.2: Message delivery and receiver-side transitions of MPI System Automaton, `MPI`

```
Infinite(T): trait
  introduces
    null:  → T
    next: T → T
  asserts sort T generated freely by null, next
```

Figure 4.3: LSL trait `Infinite` modeling MPI handles as an infinite collection of unique items.

```
sStatus(Msg): trait
  sStatus[Msg] tuple of call: sCall, msg: Msg, handle: Handle
  Request[Msg] tuple of msg: Msg, handle: Handle, complete: Bool, sent: Bool
```

Figure 4.4: LSL trait `sStatus` defining `sendStatus` and `Request` tuples.

Signature The automaton signature consists of ten action labels. Eight external actions model the call and return points for the four MPI methods. In the discussion below and in the figures, we distinguish between the four "sender" actions `Isend`, `resp_Isend`, `test`, and `resp_test` and the four "receiver" actions `Isend`, `resp_Isend`, `receive`, and `resp_receive`. The two internal actions `complete` and `move` model message delivery. We need two actions because a message may be delivered to the receiver before it is complete. (This dichotomy makes more sense in the context of more complex MPI communication patterns such as multicasts.) For compatibility of composition each action is parameterized by the channel endpoints `i` and `j`.

States The model uses two distinct data structures to maintain information about the contents of individual messages and information about the ordering of messages. The sequence `handles` maintains the order of messages while the (unordered) set `channel` collects all other information about the messages in transit including their content and status. Each message *en route* is associated with a `Request` tuple that associates the message content with a unique handle (for naming) and status information (completion, delivery). These message tuples are stored in the `channel`. The `handles` sequence simply orders the handles, thus connecting the message order to the other status information and message content. Messages are deleted from `channel` only upon a successful **test** on the message handle.

A separate sequence of messages `toRecv` is maintained. This decouples completion of **Isend**s from **receive**s. Messages are moved atomically from `channel` to `toRecv` (in the order specified by `handles`).

The `sendStatus` and `receiveStatus` tuples indicate the current method invocation(s) to which MPI is currently responding. The values of these variables include the name of the method as well as its arguments, if any. The well-formedness requirement on the client indicates that at most one receiver-method and one sender-method can be active. (See Section 4.3 for client requirements.)

Initially, `channel`, `handles`, and `toRecv` are empty and both `sendStatus` and `receiveStatus` indicate no invocations is in progress. The fields of `sendStatus` other than `sCall` may initially contain any value.

Derived Variable In addition to the state variables shown in `MPI`, it is convenient to define an auxiliary variable that is derived from the variables shown in the automaton. In our model, the `channel` set collects all the information about messages in transit except their order. The messages are kept unordered because **Isend** operations may complete in any order. However, in defining and proving the simulation relation in Section 4.7 it is also useful to describe the messages in the channel as a sequence ordered according to the handles with which they are associated.

Currently, there is no syntax defined in IOA for introducing derived variables for use in a simulation proof. As a(n ugly) workaround, we define `toSend` in the auxiliary LSL trait `toSend` shown in

Figure 4.5. Ideally, there should be notation for introducing derived variables at the beginning of a simulation relation (*e.g.*, a "let" construct) where identifiers associated with the related automata (*e.g.*, `MPI` and `channel`) are in scope. For now, we use a trait which duplicates some of the existing identifiers in order to define new ones.

In that trait, the **includes** clause references other LSL traits whose operators and axioms are required in this one. The **introduces** clauses defines four unary operators (`__.toSend`, `__.channel`, `__.handles`, and `reachable`) on the state of an `MPI` automaton. The first three operators are equivalent to state variables of the automaton. While the first is actually new, the latter two are introduced only to work around the fact that they are not automatically in scope. (Sections 11.2 and 13.2 define the correspondence between states and state variables and tuples and selection operators.) The fourth operator is a predicate asserting a state of `MPI` is reachable. See [11] and [12] for a framework for defining the semantics of such a predicate for an IOA automaton.

The **asserts** clause defines the semantics of `toSend`. In any state of `MPI`, we define the derived variable `toSend` as the sequence of messages consisting of the messages in `channel` and ordered by `handles`.

Using the `toSend` sequence, we can characterize the messages transported by `MPI` as follows. The sequence formed by the catenation of `toSend` and `toRecv` contains the messages, in order, that have been sent but not yet received while `channel` contains messages sent but not successfully tested.

```
toSend(Msg, Node): trait
  includes Sequence(Msg), sStatus(Msg), Set(Request[Msg]), Sequence(Handle)
  introduces
    __.toSend : _States[MPIAut, Msg, Node] → Seq[Msg]
    __.channel: _States[MPIAut, Msg, Node] → Set[Request[Msg]]
    __.handles: _States[MPIAut, Msg, Node] → Seq[Handle]
    reachable: _States[MPIAut, Msg, Node] → Bool
  asserts with s: _States[MPIAut, Msg, Node], i: Int, r: Request[Msg]
       reachable(s) ∧ r ∈ s.channel ⇔
         ∃ i:Int (r.msg = s.toSend[i] ∧ r.handle = s.handles[i])
```

Figure 4.5: LSL trait defining derived variable `toSend`.

Sender Transitions

`Isend(m:Msg, i, j:Int)` Initiate an **Isend**. The client promises not to access the (not modeled) memory in which `m` is stored until the send completes. Assign information about the call to `sendStatus`.

`resp_Isend(handle: Infinite, i, j:Int)` **Isend** has returned. Insert the message in `channel` and `handles`. Return a new handle to the client. The handle names this **Isend** method invocation so the client can later ask about its completion. Reset `sendStatus`.

`test(handle: Infinite, i, j:Int)` Test for completion of an **Isend**. Assign information about the call to `sendStatus`.

`resp_test(true, i, j:Int)` **test** has returned. Return `true` if the handle points to no current request or to a completed request in the channel. In the latter case, delete the request. Reset `sendStatus`.

`resp_test(false, i, j:Int)` **test** has returned. Return `false` if the handle points to an incomplete request in the channel. Reset `sendStatus`.

Message Delivery Transitions

`move(request: Request, i, j:Int)` Internal action to deliver message from `channel` to `toRecv` (in the order specified by `handles`). Atomically move a message from `channel` to `toRecv`. Replace the corresponding unsent request in `channel` with an otherwise identical sent one.

`complete(request: Request, i, j:Int)` Complete sender communication request. Any sent, incomplete request in `channel` may be replaced by an otherwise identical complete request. A completed **Isend** has the (not modeled) property that the user may reuse the memory that stores the sent message.

Receiver Transitions

`receive(i, j:Int)` Receive next available message. Assign information about the method invocation to `receiveStatus`.

`resp_receive(m:Msg, i, j:Int)` **receive** has returned. Remove first message on `toRecv` and return it. Reset `receiveStatus`.

`Iprobe(i, j:Int)` Probe for message availability. Assign information about the method invocation to `receiveStatus`.

`resp_Iprobe(flag:Bool, i, j:Int)` **Iprobe** has been invoked. Return `true` if and only if `toRecv` is not empty. Reset `receiveStatus`.

4.3 MPI Client Specification Automata

Any MPI client process that invokes a method on the MPI library will block until the method returns. (Therefore, we only use methods that are guaranteed to return "quickly"; see Section 4.1.) That blocking behavior is modeled as a handshake protocol similar to the one described in Section 3.3. We formalize that protocol by defining the MPI client specification automata shown in

Figures 4.6 and 4.7. The `MPI` automaton correctly models the behavior of MPI only if each client program composed with it implements the `SendClient` or `ReceiveClient` specifications.

Pairs of matching *call* and *return* actions model method invocation. From the client's point of view, a call action is an output that models the call point in the program. The corresponding return action is an input that models the return point. (Of course, from the point of view of `MPI` the input and output labels are reversed.) For example, the call action `Isend` represents the initiation of an **Isend** method. The return action `resp_Isend` represents the return from MPI of that method. To implement either specification, a client must not generate an output initiating a new MPI method invocation until MPI responds to any previous invocations with the appropriate `resp_*` input action.

Formally, we define a *sender trace* of an automaton to be the projection of a trace that includes only the four sender actions of the `SendClient` automaton. Similarly, a *receiver trace* of an automaton is the projection of a trace that includes only the four receiver actions of the `ReceiveClient` automaton. In every sender or receiver trace of a valid client automaton no two call actions occur consecutively. That is, a client never invokes an MPI method while an existing invocation is in progress.

There are corresponding invariants for the `MPI` automaton. In every sender or receiver trace of `MPI`, every return action is immediately preceded by the corresponding call action. That is, from the sender's point of view, MPI is passively waiting to be called and never generates an input to the client (output from MPI) unless the client asked for it (by executing a call action). Note these invariants do allow that a trace of `MPI` may arbitrarily interleave valid sender and receiver traces.

```
axioms Infinite(Handle)

automaton SendClient(M,Node:Type, i,j:Node)
  signature
    output Isend(m: M, const i, const j)
    input resp_Isend(handle:Handle, const i, const j)
    output test(handle:Handle, const i, const j)
    input resp_test(flag:Bool, const i, const j)

  states
    idle : Bool := true

  transitions
    output Isend(m,i,j)
      pre idle = true
      eff idle := false
    input resp_Isend(handle,i,j)
      eff idle := true
    output test(handle,i,j)
      pre idle = true
      eff idle := false
    input resp_test(flag,i,j)
      eff idle := true
```

Figure 4.6: `SendClient` automaton

```
automaton ReceiveClient(M,Node:Type, i, j:Node)

  signature
    output Iprobe(const i, const j)
    input resp_Iprobe(flag:Bool, const i, const j)
    output receive(const i, const j)
    input resp_receive(m: M, const i, const j)

  states
    idle : Bool := true

  transitions
    output Iprobe(i,j)
      pre idle = true
      eff idle := false
    input resp_Iprobe(flag, i,j)
      eff idle := true
    output receive(i,j)
      pre idle = true
      eff idle := false
    input resp_receive(m,i,j)
      eff idle := true
```

Figure 4.7: `ReceiveClient` automaton

4.4 Abstract Channel Specification Automaton

```
automaton AbstractChannel(Msg, Node:Type, i, j:Node)

  signature
    input SEND(m:Msg, const i, const j)
    output RECEIVE(m:Msg, const i, const j)

  states
    messages: Seq[Msg] := {}

  transitions
    input SEND(m, i, j)
      eff messages := messages ⊢ m
    output RECEIVE(m, i, j)
      pre messages ≠ {} ∧ m = head(messages)
      eff messages := tail(messages)
```

Figure 4.8: Reliable FIFO channel automaton, `AbstractChannel`

We wish to give IOA programmers the simplest possible abstract channel interface to which to connect algorithm automata and about which to reason. Therefore, we specify a one-way, point-to-point, reliable, FIFO channel as our abstraction for a communication service. Figure 4.8 details the `AbstractChannel` automaton that specifies this service. This automaton delivers messages of sort `Msg` from node `i` to node `j`. Notice that the `LCRChannel` automaton introduced in Section 2.6 is a just a specialization of the `AbstractChannel` automaton in which the `Msg` type parameter is instantiated as

Int.

The automaton is intentionally simple. The SEND action appends input messages to the queue. The RECEIVE action removes messages from the queue and outputs them in the order sent. Messages are not lost, created, duplicated, or reordered. The action names are capitalized to distinguish them from the MPI actions and to emphasize to programmers that the action names are reserved for special handling by the Interface Generator. The Interface Generator recognizes actions with these labels as the network actions of an algorithm automaton when producing buffer automata as described in Section 3.4.

4.5 Mediator Automaton

We are able to provide an IOA program the appearance of interfacing with the AbstractChannel automaton when, in fact, the program actually connects to MPI by interposing auxiliary IOA automata to mediate between MPI and the algorithm automaton. The signature of a mediator automaton must be a combination of the actions of a client automaton (to connect to MPI) and the actions of the abstract channel automaton (to connect to the algorithm automaton). In particular, the signature of the SendMediator automaton consists of all the actions of SendClient plus the SEND action from AbstractChannel. The signature of ReceiveMediator consists of all the actions of ReceiveClient plus the RECEIVE action from AbstractChannel.

4.5.1 Send Mediator Automaton

States The state of a SendMediator automaton consists of a status variable that contains one element of the enumeration Isend, test, or idle, two queues of messages toSend and sent, and a sequence of handles handles.

Messages from the user are stored in the toSend sequence. Messages are transferred from the toSend queue to the sent queue upon initiation of an Isend. Messages remain in the sent queue until the SendMediator automaton receives a true response from a test, indicating that MPI has taken responsibility for storing the message content. The handles sequence stores the handles associated with messages stored in the sent sequence.

Transitions The SEND transition stores an input message m in the toSend queue. Whenever status indicates the automaton is idle and toSend is not empty, an Isend transition may execute which outputs the message at the head of toSend, moves that message from toSend to sent, and sets status to indicate that an Isend invocation is in progress. The resp_Isend input transition resets status to idle and appends the input handle to the handles sequence.

A test transition may execute whenever status indicates the automaton is idle and handles is

```
type sCall = enumeration of idle, Isend, test

automaton SendMediator(Msg, Node:Type, i, j:Node)

  assumes Infinite(Handle)

  signature
    input  SEND(m:Msg, const i, const j)
    output Isend(m: Msg, const i, const j)
    input resp_Isend(handle:Handle, const i, const j)
    output test(handle:Handle, const i, const j)
    input resp_test(flag:Bool, const i, const j)

  states
    status: sCall := idle,
    toSend: Seq[Msg] := {},
    sent: Seq[Msg] := {},
    handles: Seq[Handle] := {}

  transitions
    input SEND(m, i, j)
      eff toSend := toSend ⊢ m
    output Isend(m,i,j)
      pre head(toSend) = m;
          status = idle
      eff toSend := tail(toSend);
          sent := sent ⊢ m;
          status := Isend
    input resp_Isend(handle, i, j)
      eff handles := handles ⊢ handle;
          status := idle
    output test(handle, i, j)
      pre status = idle;
          handle = head(handles)
      eff status := test
    input resp_test(flag, i, j)
      eff if flag then
             handles := tail(handles);
             sent := tail(sent)
          fi;
          status := idle
```

Figure 4.9: Send mediator automaton `SendMediator`

not empty. The transition outputs the handle at the head of `handle` and sets `status` to indicate that a `test` invocation is in progress. The `resp_test` input transition resets `status` to `idle` and, conditional on the truth of the `flag` input, removes the first handle and message from `handles` and `sent`, respectively. A `true` response to a `test` invocation indicates that MPI has taken responsibility for storing the message content.[4]

4.5.2 Receive Mediator Automaton

```
type rCall = enumeration of idle, receive, Iprobe

automaton ReceiveMediator(Msg, Node:Type, i, j:Node)

  assumes Infinite(Handle)

  signature
    output RECEIVE(m:Msg, const i, const j)
    output Iprobe(const i, const j)
    input resp_Iprobe(flag:Bool, const i, const j)
    output receive(const i, const j)
    input resp_receive(m: Msg, const i, const j)

  states
    status: rCall := idle,
    toRecv: Seq[Msg] := {},
    ready: Bool := false

  transitions
    output RECEIVE(m, i, j)
      pre m = head(toRecv)
      eff toRecv := tail(toRecv)
    output Iprobe(i, j)
      pre status = idle;
          ready = false
      eff status := Iprobe
    input resp_Iprobe(flag, i, j)
      eff ready := flag;
          status := idle
    output receive(i, j)
      pre ready = true;
          status = idle
      eff status := receive
    input resp_receive(m, i, j)
      eff toRecv := toRecv ⊢ m;
          ready := false;
          status := idle
```

Figure 4.10: Receive mediator automaton `ReceiveMediator`

[4] `sent` models the idea that the sender may not reuse the send buffer until a successful **test** has been performed. Moving the message from one queue to another seems to violate the spirit the restriction but it makes the proof a little easier to write. In the future, we may develop a more rigorous treatment of the memory shared between MPI and the client.

States The state of a `ReceiveMediator` automaton consists of a `status` variable that contains elements of the enumeration **receive**, `Isend`, or `idle`, a queue of messages `toRecv`, and a boolean indicator `ready`. The `ready` flag indicates that MPI has a message ready for immediate (nonblocking) receipt. Messages received from the channel are stored in the `toRecv` sequence. A message remains in `toRecv` until the `ReceiveMediator` automaton outputs it.

Transitions The `RECEIVE` transition removes the first message in the `toRecv` sequence. Whenever `status` indicates the automaton is idle and `ready` is `false`, an `Isend` transition may execute which sets `status` to indicate that an `Isend` invocation is in progress. The `resp_Isend` input transition resets `status` to `idle` and `ready` to the value of the input `flag`.

A `receive` transition may execute whenever `status` indicates the automaton is idle and `ready` is `true`. The transition sets `status` to indicate that a `receive` invocation is in progress. The `resp_receive` input transition resets `status` to `idle` and `ready` to `false` and appends the input message `m` to the sequence `toRecv`.

4.6 Composite Channel Automaton

The `CompositeChannel` automaton shown in Figure 4.11 combines our model of the MPI communication service with mediator automata to precisely define our model of the actual communication service provided to algorithm automata by the IOA compiler. The `CompositeChannel` automaton is the composition of `SendMediator`, `ReceiveMediator`, and `MPI` where all three are parameterized by the message type `Msg`, node type `Node`, and channel endpoints `i` and `j`. Notice that the endpoints of the `ReceiveMediator` automaton are reversed from those of the `MPI` automaton. All transitions except `SEND(m, i, j)` and `RECEIVE(m, j, i)` are hidden so that the external signature of `CompositeChannel` is the same as that of `AbstractChannel`. The simulation relation in that figure is the subject of the next section.

4.7 Channel Correctness Theorem

We claim the use of mediator automata allows programmers to design their systems assuming the existence of reliable, one-way FIFO channels. We now justify that claim by proving Theorem 4.1.

Theorem 4.1 The set of traces of `CompositeChannel` is a subset of the set of traces of `AbstractChannel`.

Theorem 4.1 asserts that `CompositeChannel` implements `AbstractChannel`. In other words, the former will never exhibit any externally observable behavior that could not also be exhibited by the latter. Thus, there is no way for an algorithm automaton to determine if it is connected to an `AbstractChannel` automaton as in Figure 3.1(a) or to a `CompositeChannel` automaton as in

```
axioms Infinite(Handle)

automaton CompositeChannel(Msg, Node:Type, i, j:Node)

  assumes toSend(Msg, Node)

  components
    M: MPI(Msg, Node, i, j);
    S: SendMediator(Msg, Node, i, j);
    R: ReceiveMediator(Msg, Node, j, i)

  hidden
    Isend(m, i, j), resp_Isend(handle, i, j),
    test(handle, i, j), resp_test(flag, i, j),
    receive(j, i), resp_receive(m, j, i),
    Iprobe(j, i), resp_Iprobe(flag, j, i)

forward simulation from AbstractChannel to CompositeChannel:
  if CompositeChannel.S.status = Isend then
     AbstractChannel.messages =
       CompositeChannel.R.toRecv ||
       CompositeChannel.M.toRecv ||
       (CompositeChannel.M.toSend ⊢ CompositeChannel.M.sendStatus.msg) ||
       CompositeChannel.S.toSend
  else
     AbstractChannel.messages =
       CompositeChannel.R.toRecv ||
       CompositeChannel.M.toRecv ||
       CompositeChannel.M.toSend ||
       CompositeChannel.S.toSend

  proof
    for input SEND(m:Msg, i:Int, j:Int) do
     fire input SEND(m, i, j)
    od
    for output RECEIVE(m:Msg, i:Int, j:Int) do
      fire output RECEIVE(m, i, j)
    od
    for internal Isend(m: Msg, i:Int, j:Int) ignore
    for internal resp_Isend(handle:Handle, i:Int, j:Int) ignore
    for internal test(handle:Handle, i:Int, j:Int) ignore
    for internal resp_test(flag:Bool, i:Int, j:Int) ignore
    for internal complete(request:Request, i:Int, j:Int) ignore
    for internal move(request:Request, i:Int, j:Int) ignore
    for internal receive(i:Int, j:Int) ignore
    for internal resp_receive(m: Msg, i:Int, j:Int) ignore
    for internal Iprobe(i:Int, j:Int) ignore
    for internal resp_Iprobe(flag:Bool, i:Int, j:Int) ignore
```

Figure 4.11: Composite automaton `CompositeChannel`

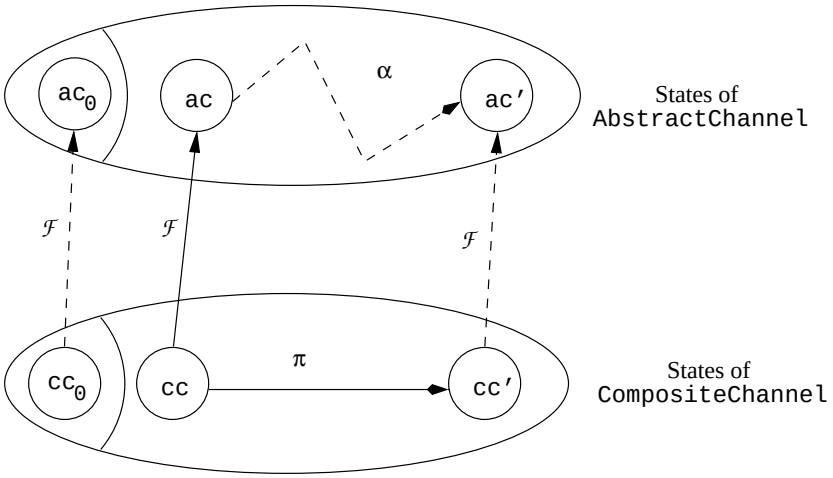

Figure 4.12: Schematic of channel refinement mapping $\mathcal{F}$. $\mathcal{F}$ is a refinement mapping from `CompositeChannel` to `AbstractChannel` if (1) for any initial state cc_0 of `CompositeChannel`, $ac_0 = \mathcal{F}(cc_0)$ is an initial state of `AbstractChannel`, and (2) for any reachable states cc and $ac = \mathcal{F}(cc)$ of `CompositeChannel` and `AbstractChannel`, respectively, and for any transition π of `CompositeChannel` enabled in state cc and resulting in state cc', there is a sequence of transitions α of `AbstractChannel` that results in state $ac' = \mathcal{F}(cc')$ where α has the same trace as π.

Figure 3.1(b). The theorem is proved by demonstrating a refinement $\mathcal{F}$ that maps the states of `CompositeChannel` to the states of `AbstractChannel`.

To be a refinement mapping, $\mathcal{F}$ is required to satisfy the following two conditions: (1) for any initial state cc_0 of `CompositeChannel`, $ac_0 = \mathcal{F}(cc_0)$ is an initial state of `AbstractChannel`, and (2) for any reachable states cc and $ac = \mathcal{F}(cc)$ of `CompositeChannel` and `AbstractChannel`, respectively, and for any transition π of `CompositeChannel` enabled in state cc and resulting in state cc', there is is (possibly empty) sequence of transitions α of `AbstractChannel` that results in state $ac' = \mathcal{F}(cc')$ where α has the same trace as π. Figure 4.12 depicts the requirements schematically.

The **forward simulation** clause in Figure 4.11 asserts the existence of just such a refinement mapping. That clause defines a refinement mapping $\mathcal{F}$ from `CompositeChannel` to `AbstractChannel`. Since `messages` is the only state variable of `AbstractChannel`, determining the value of that sequence determines a state of `AbstractChannel`. The mapping is defined by two cases. Between **Isend** invocations, the `messages` sequence in the `AbstractChannel` automaton will be the same as the catenation of the `ReceiveMediator toRecv` sequence, the `MPI toRecv` sequence, the derived `toSend` sequence, and the `SendMediator toSend` sequence ($\|$ is the catenation operator). When the `SendMediator status` variable indicates an **Isend** invocation is in progress, the message stored in the `sendStatus` tuple is included between the last two sequences.

The **proof** block in Figure 4.11 defines a step correspondence between the `CompositeChannel` automaton and the `AbstractChannel` automaton that maintains $\mathcal{F}$. That is, for every parameterized transition of `CompositeChannel` the **proof** block specifies the sequence of transitions `AbstractChannel`

can take so that the refinement mapping continues to hold. Only two of the sequences are nonempty. Those sequences (for the `SEND` and `RECEIVE` transitions) consist of the single actions of the same name in the specification automaton `AbstractChannel`. It remains to be proved that, for each transition of `CompositeChannel` and corresponding sequence of `AbstractChannel` specified in the **proof** block, $\mathcal{F}$ maps the resulting state of `CompositeChannel` to the resulting state of `AbstractChannel` and that the corresponding actions (if any) are enabled.

4.7.1 Sequence properties

The following facts about sequence partition and catenation will be referred to in the proof below.

Claim 4.2 For any sequence A, any partition $X \parallel Y$ of A such that $X = \{\}$ only if $A = \{\}$, and any element e,

1. $A \vdash e = X \parallel (Y \vdash e)$
2. $\text{tail}(A) = \text{tail}(X) \parallel Y$,
3. $\text{head}(A) = \text{head}(X)$,
4. $A = X \parallel \text{head}(Y) \parallel \text{tail}(Y)$

4.7.2 $\mathcal{F}$ is a refinement mapping

The proof that $\mathcal{F}$ is a refinement mapping from `CompositeChannel` to `AbstractChannel` proceeds by proving two lemmas. Lemma 4.3 asserts the initial state correspondence holds. Lemma 4.4 asserts a step correspondence maintains the state correspondence.

Lemma 4.3 If cc is in the set of initial states of `CompositeChannel` then $\mathcal{F}(cc)$ is in the set of initial states of `AbstractChannel`.

Proof: Let cc_0 be the unique initial state of `CompositeChannel` and ac_0 the unique initial state of `AbstractChannel`. In ac_0 `messages` is empty. In state cc_0, `S.sCall` = `idle` and `S.toSend` = `M.toSend` = `M.toRecv` = `R.toRecv` = `{}`. Thus, $\mathcal{F}(cc_0)$.`messages` = `{}` since it is just the catenation of empty sequences. Thus $\mathcal{F}(cc_0) = ac_0$, as needed. ■

Lemma 4.4 Let cc be a reachable state of `CompositeChannel`, $ac = \mathcal{F}(cc)$ be a reachable state of `AbstractChannel`, and π be a transition of `CompositeChannel` resulting in state cc'. There is an enabled sequence of transitions α of `AbstractChannel` that results in state $ac' = \mathcal{F}(cc')$ such that $trace(\alpha) = trace(\pi)$.

Proof: We prove Lemma 4.4 by case analysis of the transitions π of `CompositeChannel` and the corresponding transition sequences α of `AbstractChannel` as defined in the **proof** block of Figure 4.11. Let $ac = \mathcal{F}(cc)$ and $ac' = \mathcal{F}(cc')$.

$\pi = \alpha =$`SEND(m, i, j)` Since `SEND` is an input action, $trace(\pi) = trace(\alpha) = \pi$, as needed. The result of π is that cc'.`S.toSend` = cc.`S.toSend` $\vdash$ `m` while all other state variables in cc' are unchanged from cc. Since $ac = \mathcal{F}(cc)$, cc.`S.toSend` is a suffix of ac.`messages` (in both cases of $\mathcal{F}$). Thus, we apply Claim 4.2.1 by letting A be ac.`messages`, Y be cc.`S.toSend`, and e be `m` to conclude that appending `m` to cc.`S.toSend` is equivalent to appending `m` to ac.`messages`. Thus, ac'.`messages` = ac.`messages` $\vdash$ `m` which is exactly the state produced by α from state ac, as needed. Finally, since `SEND` is an input action, α is enabled ac, as needed.

$\pi = \alpha =$`RECEIVE(m, i, j)` Since `RECEIVE` is an output action, $trace(\pi) = trace(\alpha) = \pi$, as needed. The result of π is that cc'.`R.toRecv` = tail(cc.`R.toRecv`) while all other state variables in cc' are as in cc. Since $ac = \mathcal{F}(cc)$, cc.`R.toRecv` is a prefix of ac.`messages` (in both cases of $\mathcal{F}$). We apply Claim 4.2.2 by letting A be ac.`messages` and Y be cc.`R.toRecv` to conclude that tailing cc.`R.toRecv` is equivalent to tailing ac.`messages`. Thus, ac'.`messages` = tail(ac.`messages`) which is exactly the state produced by α from state ac, as needed.

It remains to show that α is enabled in state ac. That is, we must show that the precondition for `RECEIVE` is `true` in ac. Thus, it is sufficient to show that `m` = head(ac.`messages`). Since π is enabled in cc, it follows that `m` = head(cc.`R.toRecv`). We apply Claim 4.2.3 by letting Y be ac.`messages` and X be cc.`R.toRecv` to conclude that `m` = head(ac.`messages`), as needed.

$\pi =$ `Isend(m, i, j)`, $\alpha = \{\}$ Since `Isend` is hidden, $trace(\pi)$ is empty, as needed. Since π is enabled, cc.`S.status` = `idle`. A result of π is that cc'.`S.sCall` = `Isend`. Since `S.sCall` changes from `idle` to `Isend`, $ac = \mathcal{F}(cc)$ is defined by the **else** case of the state correspondence while $ac' = \mathcal{F}(cc')$ is covered by the first case. Since $ac = \mathcal{F}(cc)$, cc.`S.toSend` is a suffix of ac.`messages`. Furthermore, cc'.`S.toSend` = tail(cc.`S.toSend`), and cc'.`M.sendStatus.msg` = `m`. Since π is enabled, `m` = head(cc.`S.toSend`). We apply Claim 4.2.4 by letting A be ac.`messages` and Y be cc.`S.toSend` to conclude that moving the first message in cc.`S.toSend` to the end of cc.`M.toSend` does not change ac.`messages`. Since cc'.`M.sendStatus.msg` is appended to the end of cc'.`M.toSend` in the `Isend` case, ac'.`messages` = ac.`messages`, as needed.

$\pi =$ `resp_Isend(handle, i, j)`, $\alpha = \{\}$ Since `resp_Isend` is hidden, $trace(\pi)$ is empty, as needed. Since π is enabled, cc.`S.status` = `Isend`. A result of π is that cc'.`S.status` = `idle`. Since `S.sCall` changes from `Isend` to `idle`, $ac = \mathcal{F}(cc)$ is defined by the first case of the state correspondence while $ac' = \mathcal{F}(cc')$ is covered by the **else** case. Since π is enabled, `handle` = `Request.handle`. Thus, another result of π is that cc'.`M.handles` = cc'.`M.handles` $\vdash$ `Request.handle`. Furthermore,

cc'.M.channel contains a new request pairing handle with cc.M.sendStatus.msg. Thus, by the definition of toSend, cc'.M.toSend = cc.M.toSend $\vdash$ cc.M.sendStatus.msg. However, appending cc.M.sendStatus.msg to cc.M.toSend is exactly the difference between the Isend and **else** cases of the state correspondence. Thus, ac.messages = ac'.messages, as needed.

π = test(handle, i, j), $\alpha = \{\}$ Since test is hidden, $trace(\pi)$ is empty, as needed. Since π is enabled, cc.S.status = idle. A result of π is that cc'.S.status = test. Thus, the same case of $\mathcal{F}$ applies to both cc and cc'. Since π does not modify any queues mentioned in the state relation, $\mathcal{F}(cc) = \mathcal{F}(cc')$ and ac.messages = ac'.messages, as needed.

π = resp_test(flag, i, j), $\alpha = \{\}$ Since resp_test' is hidden, $trace(\pi)$ is empty, as needed. Since π is enabled, cc.S.status = test. A result of π is that cc'.S.status = idle. Thus, the same case of $\mathcal{F}$ applies to both cc and cc'. Since π does not modify any queues mentioned in the state correspondence, $\mathcal{F}(cc) = \mathcal{F}(cc')$ and ac.messages = ac'.messages, as needed.

π =move(request, i, j), $\alpha = \{\}$ Since move is internal, $trace(\pi)$ is empty, as needed. Since π does not effect cc.S.status, the same case of $\mathcal{F}$ applies to both cc and cc'. Since π is enabled, Request.handle = head(cc.M.handles) and Request.handle $\in$ cc.M.channel. Therefore, by the definition of toSend, Request.msg = head(cc.M.toSend). Since cc'.M.handles = tail(cc.M.handles) and no requests are deleted from cc.M.channel in this transition, cc'.M.toSend = tail(cc.M.toSend). Furthermore, cc'.M.toRecv = cc.M.toRecv $\vdash$ Request.msg. We apply Claim 4.2.4 by letting A be ac.messages and X be the prefix of ac.messages up to and including cc.M.toSend and Y be the suffix of ac.messages starting at cc.M.toSend to conclude that moving the first message in cc.M.toSend to the end of cc.M.toRecv does not change ac.messages. Since that move is exactly the difference between cc and cc', ac.messages = ac'.messages, as needed.

Note, since cc.M.toSend is not empty and the suffix changes only by the deletion its first element, the case of $\mathcal{F}$ that applies is irrelevant.

π =complete(request, i, j), $\alpha = \{\}$ Since complete is internal, $trace(\pi)$ is empty, as needed. Since π does not effect cc.S.status, the same case of $\mathcal{F}$ applies to both cc and cc'. The result of π is to replace one incomplete request in cc.M.channel with an otherwise identical complete one. The definition of toSend, and, thus the definition of $\mathcal{F}$ is not sensitive to the substitution. Therefore, ac.messages = ac'.messages, as needed.

π =receive(i, j), $\alpha = \{\}$ Since receive is hidden, $trace(\pi)$ is empty, as needed. Since π does not effect cc.S.status, the same case of $\mathcal{F}$ applies to both cc and cc'. Since π does not modify any queues mentioned in the state relation, $\mathcal{F}(cc) = \mathcal{F}(cc')$ and ac.messages = ac'.messages, as needed.

π =`resp_receive(m,i, j)`, $\alpha = \{\}$ Since `resp_receive` is hidden, $trace(\pi)$ is empty, as needed. Since π does not effect cc.`S.status`, the same case of $\mathcal{F}$ applies to both cc and cc'. Since π is enabled, head(cc.`M.toRecv`) = `m`. A result of π is that cc'.`M.toRecv` = tail(cc.`M.toRecv`). We apply Claim 4.2.4, by letting A be ac.`messages` and Y be cc.`M.toRecv` to conclude that moving the first message in cc.`M.toRecv` to the end of cc.`R.toRecv` does not change ac.`messages`. Since that move is exactly the difference between cc and cc', ac.`messages` = ac'.`messages`, as needed.

π =`Iprobe(i, j)`, $\alpha = \{\}$ Since `Isend` is hidden, $trace(\pi)$ is empty, as needed. Since π does not affect cc.`S.status`, the same case of $\mathcal{F}$ applies to both cc and cc'. Since π does not modify any queues mentioned in the state relation, $\mathcal{F}(cc) = \mathcal{F}(cc')$ and ac.`messages` = ac'.`messages`, as needed.

π =`resp_Iprobe(flag,i, j)`, $\alpha = \{\}$ Since `resp_Isend` is hidden, $trace(\pi)$ is empty, as needed. Since π does not affect cc.`S.status`, the same case of $\mathcal{F}$ applies to both cc and cc'. Since π does not modify any queues mentioned in the state relation, $\mathcal{F}(cc) = \mathcal{F}(cc')$ and ac.`messages` = ac'.`messages`, as needed.

■

4.8 Other network services

The particular choice to use MPI and provide a reliable FIFO channel service to IOA programmers is orthogonal to the basic design of the IOA compiler. The approach we use with MPI could be used to either connect compiled programs to a different network service (*e.g.*, TCP [101], JMS [117], or RMI [116]) or to change the target abstract channel interface or semantics (*e.g.*,, a lossy or broadcast channel). To do so, we would follow the same four step process. First model the network service as an IOA automaton. Second, specify the desired abstract channel as an IOA automaton. Third, write mediator automata such that the composition of the mediator automata and the external service automaton implements the abstract channel automaton. Fourth, prove that implementation relationship. In addition, as explained in Chapter 6, the compiler treats MPI transitions as special cases when emitting Java, so these special cases would also have to be updated to account for the interface to the new service.

Chapter 5

Resolving Nondeterminism

> *"Would you tell me, please, which way I ought to go from here?" "That depends a good deal on where you want to get to," said the Cat. "I don't much care where–" said Alice. "Then it doesn't matter which way you go," said the Cat.*
>
> — Lewis Carroll [16]

The IOA language is inherently nondeterministic. An automaton may start in any of a set of states. In any state an automaton reaches during execution, many actions may be enabled and, for any enabled action, several following states may result. Developing a method to resolve nondeterminism was the largest conceptual hurdle in the initial design of an IOA compiler to translate programs written in precondition-effect style IOA code into an imperative language like Java. The process of resolving nondeterminism in an IOA program is called *scheduling* the program. Before we can compile an IOA specification of a distributed system, we must resolve both the implicit nondeterminism inherent in any IOA program and any explicit nondeterminism introduced by the programmer in **choose** statements. Our approach to both parts of scheduling is the same: we let the programmer do it.

5.1 Scheduling

While any IOA program has only a finite set of transition definitions, they may be parameterized by variables of infinite types (*e.g.*, integers). Picking a transition to execute includes picking both a transition definition and the values of its parameters. It is possible and, in fact, common that the set of enabled actions in any state is infinite. It is also possible for this set to be undecidable because,

transition preconditions may be arbitrary predicates in first-order logic. Thus, there is no simple search method for finding an enabled action. Even if all transition preconditions are decidable, the general problem of determining, for a given state of a given automaton, whether or not there exists a tuple of a transition definition and a set of parameter values that will enable that transition in that state may require enumerating all tuples of transition definitions and parameters. Since common parameter sorts (*e.g.*, `Int` or `Seq[Nat]`) are countably infinite, this problem is recursively enumerable.

One might imagine that by restricting the class of automata accepted for compilation one could practically employ a solution based on exhaustive search to solve the scheduling problem. However, it is not obvious how to formulate such restrictions without also radically restricting the expressive power of the language. In any case, the human-based scheduling solution has the additional advantage of relieving the compiler designer of the burden of finding a *good* schedule, *i.e.*, one that makes actual progress rather than merely executing any enabled action.

Therefore, before compilation, we require that the programmer write a schedule. A schedule is a function of the state of the local node that picks the next action to execute at that node. In format, a schedule is written at the IOA level in an auxiliary *nondeterminism resolution language* (NDR) consisting of imperative programming constructs similar to those used in IOA effects clauses. An NDR **schedule** block encapsulates all the scheduling information for an automaton. In addition, to common control structures such as assignments, conditionals, and loops, NDR supports a **fire** statement. The NDR **fire** statement causes a transition to run and selects the values of its parameters. Schedules may reference, but not modify, automaton state variables. However, schedules may declare and modify additional variables local to the schedule. The NDR language was designed and implemented by Antonio Ramírez-Robredo and modified by Laura Dean to describes schedules for automata interpreted by the IOA simulator [102, 30]. Michael Tsai extended and reimplemented the language for use by the IOA compiler [119].

5.1.1 LCR Schedule

Figure 5.1 shows an NDR schedule block that can be used to schedule the `LCRNode` automaton. The schedule is essentially round robin in nature; it tries each transition definition in turn. The schedule loops continuously until the automaton announces itself leader.[1] The **fire** statement for each transition and its guard are derived (by hand) from the transitions precondition and **where** clauses. In most cases, the derivation includes the substitution of `MPIrank` for the parameter naming this node and `mod(MPIrank ± 1, MPIsize)` for its right/left neighbor. Other values are filled in as needed.

[1]This version of the LCR algorithm only terminates at the leader. Termination at every node can be achieved using an extra communication round in which the leader sends a message around the ring announcing its status. In that version, node schedules terminate after the announcement message has been forwarded and the sending of the forwarding message successfully tested. We use such a modified verion of LCR for experiments in Chapter 8.

```
schedule
do
  while P.status ≠ announced do
    if P.status = elected then fire output leader(MPIrank) fi;
    if len(I.stdin) > 0
       ∧ head(I.stdin).action = vote
       ∧ len(head(I.stdin).params) = 1
       ∧ tag(head(I.stdin).params[0]) = Int
       ∧ head(I.stdin).params[0].Int = MPIrank then
       fire internal vote(MPIrank)
    fi;
    if P.status ≠ idle ∧ size(P.pending) ≠ 0 then
       fire internal SEND(chooseRandom(P.pending), MPIrank,
                          mod(MPIrank + 1, MPIsize))
    fi;
    if SM[mod(MPIrank + 1, MPIsize)].status = idle ∧
       SM[mod(MPIrank + 1, MPIsize)].toSend ≠ {} then
       fire output Isend(head(SM[mod(MPIrank + 1, MPIsize)].toSend),
                         MPIrank, mod(MPIrank + 1, MPIsize))
    fi;
    if SM[mod(MPIrank + 1, MPIsize)].status = idle ∧
       SM[mod(MPIrank + 1, MPIsize)].handles ≠ {} then
       fire output test(head(SM[mod(MPIrank + 1, MPIsize)].handles),
                        MPIrank, mod(MPIrank + 1, MPIsize))
    fi;
    if RM[mod(MPIrank - 1, MPIsize)].status = idle ∧
       RM[mod(MPIrank - 1, MPIsize)].ready = false then
       fire output Iprobe(MPIrank, mod(MPIrank - 1, MPIsize))
    fi;
    if RM[mod(MPIrank - 1, MPIsize)].status = idle ∧
       RM[mod(MPIrank - 1, MPIsize)].ready = true then
       fire output receive(MPIrank, mod(MPIrank - 1, MPIsize))
    fi;
    if RM[mod(MPIrank - 1, MPIsize)].toRecv ≠ {} then
       fire internal RECEIVE(head(RM[mod(MPIrank - 1, MPIsize)].toRecv),
                             mod(MPIrank - 1, MPIsize), MPIrank)
    fi
  od
od
```

Figure 5.1: NDR schedule block for the `LCRNode` node automaton

The parameters to the **fire** statements select values that we know will satisfy the transition precondition. For example, the guard to the **fire** statement for the `Isend` transition requires that the `SendMediator` for the channel to the node's right hand neighbor is idle and that some message is waiting to be sent. The **fire** statement specifies that the message parameter `m` is the head of the `toSend` sequence for that right hand `SendMediator` as required by the precondition. The nodes parameter `N12` and `N13` are bound to `MPIrank` and `mod(MPIrank + 1, MPIsize)`, respectively, as required by the action **where** clause.

```
ChoiceMset(E): trait
  % A set with a choose operator specified to be random
  includes Multiset(E)
  introduces
    chooseRandom:  Mset[E] → E
    rest:    Mset[E] → Mset[E]
    isEmpty: Mset[E] → Bool
  asserts with e, e1: E, s: Mset[E]
    s ≠ {} ⇒ chooseRandom(s) ∈ s;
    s ≠ {} ⇒ rest(s) = delete(chooseRandom(s), s);
    s ≠ {} ⇒ s = insert(chooseRandom(s), rest(s));
    isEmpty(s) ⇔ s = {}
```

Figure 5.2: Trait `ChoiceMset` defining the `chooseRandom` operator on multisets

The trait `ChoiceMset` shown in Figure 5.2 introduces and defines the `chooseRandom` operator used in the schedule. The operator is used to produce an arbitrary (but not actually random) message in the `pending` multiset to `SEND`.[2] In the scheduled node automaton submitted for compilation, the trait is referenced using the statement **axioms** `ChoiceMset(Int)`.

5.1.2 Schedule actions

Adding an NDR schedule to an IOA program is equivalent to applying a strong form of the next-action determinism (NAD) transformation of IOA programs described by Vaziri *et al.* [120]. An I/O automaton is *next-action deterministic* if there is at most one enabled action in all reachable states. The NAD transformation produces such an automaton by adding a nondeterministic internal schedule action to an automaton. Adding an NDR schedule is similar but adds a schedule action with a deterministic and computable effect.

Conceptually, the NDR schedule adds new variables, modifies each transition to use the new variables, and adds a special schedule action. The new state variables consist of a program counter (PC), additional variables to represent transition parameters, a special schedule history variable, and whatever variables the programmer uses in the NDR schedule program. Each locally-controlled action is modified in two ways. First, the precondition is strengthened so that the action is enabled

[2] All LSL operators are total functions. Thus, despite its name, the `chooseRandom` operator always returns (and the `rest` operator always omits) the *same* element of a given set. The trait does not specify *which* element of the set is returned (or omitted).

only if the PC names the action and the transition parameters equal the corresponding new parameter state variables. Second, at the end of the effects the PC is assigned to point to the special schedule action.

The added schedule action has no parameters. The precondition of the schedule requires the PC to name the schedule action. The effect of the schedule is to compute and assign new values to the PC, the new parameters state variables, the PC history variable, and the programmer provided schedule variables. These new values are computed as a function of the augmented state. The schedule history variable allows the schedule to maintain state between executions. This allows the next invocation of the schedule to begin execution at the point immediately following the last executed **fire** statement.

5.2 Choosing

As mentioned in Section 2.6, in addition to the implicit scheduling nondeterminism in IOA, the **choose** statement introduces explicit nondeterminism. When a **choose** statement is executed, an IOA program selects an arbitrary value from a specified set. For example, the statement

```
num := choose n:Int where 0 ≤ n ∧ n < 3
```

assigns either 0, 1, or 2 to `num`. As with finding parameterized transitions to schedule, finding values to satisfy the **where** predicates of **choose** statements is, in general, hard. So, again, we require the IOA programmer to resolve the nondeterminism before compilation. In this case, the programmer annotates the **choose** statement with a *determinator block* written in the NDR language. The NDR **yield** statement specifies the value to resolve a nondeterministic choice. Determinator blocks may reference, but not modify, automaton state variables. For the above example, a determinator block might be

```
num := choose n:Int where 0 ≤ n ∧ n < 3
          det  do yield 1; yield 2; yield 2 od
```

This determinator block will assign 1 to `num` the first time this assignment executes and 2 the next two times. Further executions of the statement will repeat this cycle of three values. Using the NDR program, 0 is never assigned to `num`. Notice that determinator blocks implicitly have state. Like CLU iterators [78], a determinator block remembers the last **yield** statement executed. Thus, each determinator block also adds a PC variable to the automaton.

5.3 Initialization

The execution of an I/O automaton may start in any of a set of states. In an IOA program, there are two ways to denote its start states. First, each state variable may be assigned an initial value. That initial value may be a simple term or an explicit choice. In the latter case, the choice must

be annotated with a choice **det** block to select the initial value before code generation. Second, the initial values of state variables may be collectively constrained by an **initially** clause. As with preconditions, an **initially** clause may be an arbitrary predicate in first order logic. Thus, there is no simple search method for finding an assignment of values to state variables to satisfy an **initially** clause.

The original design for the IOA compiler required that an IOA program submitted for compilation not have an **initially** clause. However, that requirement was dropped after development of the specification for the expansion of the states of composite automata that constitutes Part II of this thesis. In that part, we specify that an expanded automaton contains an **initially** clause whenever a component automaton has non-type parameters (even if the component itself has no **initially** clause).

Therefore, in the current design, we allow **initially** clauses in automata submitted for compilation but we require the IOA programmer to annotate the **initially** predicate with a determinator block. However, unlike NDR programs for automaton schedules and **det** blocks for resolving choices, **initially det** blocks may assign values directly to state variables. (In fact, such assignments are the only way for the **det** block to set the initial state variable values.)

```
det do
    P := [{name}, idle];
    RM := update(empty, mod(MPIrank-1,MPIsize), [idle, {}, false]);
    SM := update(empty, mod(MPIrank+1,MPIsize), [idle, {}, {}, {}]);
    I := [{}, {}]
od
```

Figure 5.3: NDR **initially det** block for the `LCRNode` node automaton

The NDR **initially det** block for the LCR node automaton is shown in Figure 5.3. The block consists of four assignments; one to each of tuples derived from each component. The assignments result in a state that satisfies the **initially** clause in for the expanded automaton shown in Appendix A.1.

5.4 Safety

It is important to note that schedules and determinator blocks do not change the semantics of IOA programs but rather just annotate them with mechanisms to resolve nondeterminism. Furthermore, a schedule or determinator block is allowed to examine only the state of the single automaton that it annotates.

The programmer writes an IOA program and can then prove formally that it performs only safe behaviors. NDR annotations are responsible only for resolving nondeterminism. The role of an NDR block is to produce a value (or set of values) that satisfies some known predicates. An NDR schedule must produce a transition together with its parameters that satisfies the conjunction of the action

and transition **where** clauses and the transition precondition. Choice determinator blocks must produce a value that satisfies the **choose where** clause. Determinator blocks for **initially** clauses must produce values such that all state variables satisfy the clause.

To ensure the automaton only executes safe schedules, these conditions can be checked after the value is produced. In fact, the IOA compiler generates code to check at runtime that transition values and parameters do, in fact, satisfy the precondition and **where** clauses. Generating a check for the other cases would be analogous.

Unfortunately, we cannot verify arbitrary predicates. When **where** clauses and preconditions include universal quantifiers (particularly over infinite types) there is no easy way to emit code to verify them. Therefore, only predicates that do not contain quantifiers are currently checked. However, as described in Section 15.5, the **initially** clause generated during the expansion of a composite automaton includes universal quantifiers. Therefore, no check is currently generated for **det** blocks for **initially** clauses. Finally, while NDR schedules can be checked using the IOA simulator, there is currently no support in the toolkit for translating NDR constructs for use in mechanically checked proofs.

Chapter 6

Translating IOA into Java

Once you understand how to write a program get someone else to write it.

— Alan J. Perlis [99]

The IOA compiler is applied to each node automaton individually to produce a single Java class named for the source node automaton. The generated class subclasses a generic automaton class in our standard compiler libraries. These standard libraries also include support for console interactions, MPI initialization, automaton parameter initialization, and implementations for the standard IOA datatypes. At run time, the node automaton subclass must be linked with those standard libraries, an MPI library, and any additional implementation classes for special datatypes provided by the programmer. The automaton class is organized around a main loop derived from the NDR schedule annotation to the IOA program. Two other threads process input and output actions, placing them in the appropriate buffers as they arrive.

Below we discuss the elements of the generated automaton subclass. Section 6.1 discusses the translation of IOA sorts and operators into Java. Section 6.2 discusses the state variables of the generated program and how they are initialized. Section 6.3 discusses initialization of automaton parameters. Section 6.4 discusses how the compiler translates the parts of a transition into Java constructs including the special cases of MPI and buffer transitions. Section 6.5 discusses the translation of the main loop and the initialization **det** block.

6.1 Translating Datatypes

IOA has been designed to work closely with the Larch Shared Language (LSL) [52, 51]. All datatypes used in IOA programs are described formally in LSL. These specifications give axiomatic descriptions

of each datatype and its operators in first-order logic. While such specifications provide sound bases for proofs, it is not easy to translate them automatically into an imperative language such as Java.

In our prototype compiler, we require the programmer to write datatype implementation classes by hand. Each datatype (*e.g.*, `Bool`, `Int`, or `Mset[__]`) is implemented as a Java class. Each operator (*e.g.*, `__>__: Int, Int → Bool` or `size: Mset[__] → Int`) is implemented as a method by some datatype implementation. Since operators often have more than one type in their signature, it is not obvious with which datatype to associate an operator. The IOA compiler relies on guidance from the datatype implementor to match IOA operators and datatypes to the corresponding Java methods and classes. This guidance is provided in the form of a *registration class* associated with each datatype implementation class. The registration class tells the IOA compiler which implementation class is associated with each datatype and operator. The mapping between datatypes and operators and implementation classes and methods is maintained in a datatype registry [119, 121].

The IOA toolkit includes a standard library of implementation classes for the standard language datatype. These include simple datatypes such as naturals, integers, and booleans, compound datatypes such as arrays, maps, sets, and sequences, and shorthand types such as enumerations, tuples, and unions. Programmers are free to extend the compiler with new datatypes or replace the standard implementations with their own. (Instructions are provided in [119] for how to do so.) For example, the compiler can be configured to replace the standard implementation class for `Int` based on the bounded type **java.lang.Integer** with one based on the (nearly) unbounded type **java.math.BigInteger**. The programmer specifies at compile time which datatypes to load, and the datatype registry is initialized appropriately [94].

Since the IOA framework focuses on correctness of the concurrent, interactive aspects of programs rather than of the sequential aspects, we do not address the problem of establishing the correctness of this sequential code (other than by conventional testing and code inspection). Standard techniques of sequential program verification (based, for example, on Hoare logic) may be applied to attempt such correctness proofs.

6.2 Translating State

Each state variable of the IOA program is translated into a member variable of the generated Java automaton class. The classes implementing the types of these variables must either be included in the standard datatype library or provided by the programmer. If the IOA variable is assigned a particular initial value, that value is translated like any other IOA term into a Java value and assigned to that value before any transitions are run. If no initial value is specified in the IOA program then the initial value is, implicitly, an unconstrained **choose** value. That is, the variable may be arbitrarily initialized to any value of the correct datatype.

Therefore, implementation classes for simple sorts are required to have a method to construct *some* instance of that datatype. The emitted code calls that construction method at runtime to initialize member variables of that sort.

If the datatype is a constructed sort such as `Array[Int, Bool]`, the process is slightly more complicated. Currently, every constructed sort is implemented by a single implementation class in the standard datatype library. For example, every IOA `Array`, without regard to its subsorts (*i.e.*, to its sort parameters), is implemented by the single class **ioa.runtime.adt.ArraySort**. (This is not required by the datatype registry design. It is possible to register different implementation classes for arrays containing or indexed by different types.) Furthermore, a constructed sort usually contains some instances of its subsorts. For example, an `Array[Int, Bool]` must contain a boolean value in every entry while an `Array[Int, Mset[Nat]]` must contain a multi-set of naturals. Even when no instance of a subtype needs to be constructed (say, for an empty set), it may be necessary to know from which subsorts a particular sort was constructed in order to make an instance of the appropriate datatype.

Therefore, the code emitted to initialize a member variable of a constructed sort must contain all the information necessary to construct instances of the implementation classes of the subsorts. At runtime, the constructed type traverses this information (called a **Parameterization**) to recursively construct all the needed instances of the the implementation classes of the subsorts. Tsai's thesis details the construction and use of **Parameterization**s [119].

6.3 Translating Automaton Parameters

If the source automaton is parameterized, additional member variables are generated to represent those parameters. The values of automaton parameters are determined only at runtime. For most parameters, the initial value is read from a per-node input file. Thus, the end user can specify distinct values for each node in the system.

Two parameters are handled as special cases. If the automaton has a parameter named `MPIrank` whose sort is implemented by a class that implements the **ioa.runtime.adt.MPINode** interface, then the IOA compiler not only generates a member variable for that parameter, it also generates code that initializes the variable with the value returned by the **MPI.COMM_WORLD.Rank** method call after initialization of the MPI communications system. (This interface requires that a class be able to construct an instance from a positive Java int and to return the value from which an instance was constructed.) Both **ioa.runtime.adt.IntSort** and **ioa.runtime.adt.NatSort** (the standard implementation classes for the built-in IOA sorts `Int` and `Nat`, respectively) implement this interface. The rank of an MPI process is unique within the system. Similarly, if the automaton has a parameter named `MPIsize` then the corresponding member variable is initialized to contain

the number of nodes in the system as returned by the **MPI.COMM_WORLD.Size** method. The values of the parameters do not change after initialization because no statement in the source automaton can assign to automaton parameters.

6.4 Translating Transitions

An IOA transition definition consists of a list of parameters, a **where** clause, a precondition, and effects. The **where** clause restricts the values of the parameters. The precondition is a predicate on automaton state variables and the transition parameters that specifies when the transition is enabled. The effects specify how state variables change.

The transition **where** clause and precondition are translated into Java boolean expressions. These expressions are evaluated at run time after a schedule specifies the transition to **fire**. If the precondition or action or transition **where** clauses evaluates to false, the transition is not executed and a runtime exception is thrown. If they all evaluate to true, the effects clause is executed.

The effects clause is translated to a Java method. The basic control structures of IOA have direct analogues in Java. Thus, IOA assignments, conditionals, and loops are translated into Java assignments, conditionals, and loops. (Technically, any IOA loop can be implemented by a Java loop but the reverse is not true.) IOA conditionals are exactly analogous to their Java counterparts in syntax and semantics. So the translation of conditionals is completely straightforward. We discuss the remaining control structures below.

Translating Assignments In most cases, an assignment statement consists of a variable on the left hand side (LHS) and some term on the right hand side (RHS). Translating such assignments to Java is completely straightforward. (IOA **choose** statements are compiled by translating their associated NDR determinator blocks. While not included in the current prototype, the translation should also include runtime checks that the value produced satisfies the **choose** predicate.) However, the LHS may consist of a term more complicated than a simple variable reference if the top level operator in the term is one of three special cases.

Usually, the meaning of IOA syntax denoting an operator is independent of the context in which it appears. The only three exceptions to that rule are the syntactic structures denoting array indexing, map indexing, and tuple selection. In these cases, the syntax appearing on the LHS of an assignment indicates a location in which to store a new value, an lvalue. The same syntax appearing anywhere else indicates the current value stored in that element of the array or map or in that field of the tuple. Semantically, identical syntax indicates different operators.

If `A` is an array indexed by sorts `I1`, `I2`, ... and storing elements of sort `E`, and `i1`, `i2`, ... are terms of sort `I1`, `I2`, ..., respectively, then the syntax `A[i1, i2, ...]` appearing anywhere but the LHS of an assignment indicates the application of the array reference operator `__[__,__,...]` :

ASSIGNMENT FORM	REWRITTEN AS
`A[i1, i2, ...] := RHS`	`A := assign(A, i1, i2, ..., RHS)`
`M[i1, i2, ...] := RHS`	`M := update(M, i1, i2, ..., RHS)`
`T.f := RHS`	`T := set_f(T, RHS)`

Table 6.1: Rewrite rules to desugar assignments to arrays, maps, and tuples.

`Array[I1, I2, ..., E]` → `E` to `A`, `i1`, `i2`, ... However if the same syntax appears on the LHS of an assignment such as

```
A[i1, i2, ...] := e
```

where `e` is a term of sort `E`, then the operator is interpreted as the array assignment operator with signature `assign : Array[I1, I2, ..., E], I1, I2, ..., E → Array[I1, I2, ..., E]`. Thus, the assignment statement is semantically equivalent to:

```
A := assign(A[i1, i2, ...], e).
```

Similar interpretations apply to `Map` and **tuple** lvalues. For a `Map M`, the brackets in the syntax `M[i1, i2, ...]` usually denotes the map reference operator; but on the LHS of an assignment, the brackets are interpreted as the operator `update : Map[I1, I2, ..., E], I1, I2, ..., E → Map[I1, I2, ..., E]`.

For a **tuple** `t` with field `f` of sort `F`, the dot in the syntax `t.f` usually denotes the tuple field reference operator `get_f: T → E`; but on the LHS of an assignment, the dot operator is interpreted as `set_f: T, E → T`.

In the code generator, these correspondences are applied as rewrite rules. Each of the former forms are rewritten as assignments of the latter form before translation. That is, the syntax allowing operators on the LHS of an assignment are treated as syntactic sugar for the longer forms. The rewrite rules are summarized in Table 6.1.

The matter is slightly complicated by the fact that such lvalues may be nested. We have already seen such nested assignments in the expanded version of `LCRNode`. For example, the first statement of the effect of the `SEND` transition is the assignment:

```
SM[mod(MPIrank + 1, MPIsize)].toSend :=
              (SM[mod(MPIrank + 1, MPIsize)].toSend) ⊢ m;
```

(see Appendix A.1). Recall that `SM` is a `Map` from `Int` to `_States[SendMediator, Int, Int]` and that the latter is a **tuple**.

Thus, we must apply the third and then the second rewrite rules from Table 6.1 to convert the assignment to

```
SM[mod(MPIrank + 1, MPIsize)] :=
              set_toSend(SM[mod(MPIrank + 1, MPIsize)],
                         (SM[mod(MPIrank + 1, MPIsize)].toSend) ⊢ m);
```

and then to

```
SM := update(SM,
```

```
mod(MPIrank + 1, MPIsize),
set_toSend(SM[mod(MPIrank + 1, MPIsize)],
           (SM[mod(MPIrank + 1, MPIsize)].toSend) ⊢ m));
```

for proper translation to Java.

Translating Loops Loops may be defined in IOA using two different syntactic forms:

```
for v:T in s:Set[T] do ... od;
for v:T where P(v):Bool do ... od
```

The semantics of the first form are that the statements in the **do ... od** block are executed `size(s)` times with the variable `v` taking on a distinct value in the set `s` on each iteration.[1] The semantics of the second form are similar but the set of values `v` may take on are defined implicitly. The set consists of all values of `v` that satisfy the predicate `P`. (Here the notation `P(v)` is used to denote an arbitrary IOA term in which `v` may appear.)

Loops of the first form are supported when the implementation class for the IOA `Set` constructor implements the **ioa.runtime.adt.Enumerable** interface. Such classes can produce iterators over the elements in the `Set`. The implementation class for `Set` in the standard datatype library implements the interface.

Since the loop **where** clause may be an arbitrary predicate, implementing loops of the second form would be analogous to solving the scheduling problem discussed in Chapter 5. Currently, the IOA compiler cannot translate loops of the second form. In the future, such loops might be translated with the help of an additional NDR loop scheduling block.

Translating det blocks Each determinator block translates into a Java method named **resolveChoose**N (where N is a unique identifier for the block). In addition, the compiler generates an instance variable **choosePC**N. When a **choose** value is required to complete an assignment the corresponding method **resolveChoose** is invoked to generate that value.

As designed and implemented by Tsai [119] (based on the simulation of NDR constructs by Ramırez-Robredo [102]), the body of the resolution method is a Java **switch** statement on the **choosePC** N variable inside an infinite **while** loop. Primitive statements (assignments, conditionals, loops, and **yield**s) translate into a case or set of cases in the switch. The last statement of each block sets the program counter to the number of the next statement to execute. In the body of these cases, assignments and terms are translated just like those appearing in an IOA **eff** clause. Conditional and loop control structures are flattened into sequences of cases.

In essence, the "NDR program is translated to a Java implementation of machine code" where each "NDR statement is assigned an address, and a program counter variable keeps track of the current execution point in the NDR program." [119] The method returns when a **yield** statement

[1] IOA requires that the result of executing the loop for any ordering of values in `s` must always be the same.

is executed. The PC value however, is maintained so that the next time the **det** block executes, it picks up where it left off, implementing the required CLU-iterator-like semantics.

6.4.1 Translating MPI Transitions

As described in Chapter 4, the IOA interface to MPI is specified as a set of four pairs of transition definitions. The definition of these transitions is fixed by the mediator automata `sendMediator` (see Figure 4.9) and `recvMediator` (see Figure 4.10). These pairs of transitions are designed to mirror the corresponding four Java calls used to invoke MPI (**Isend**, **test**, **Iprobe**, and **receive**). While the IOA transitions contain all the necessary information to model the MPI method invocations, the IOA code does not translate directly using the standard mechanisms.

Instead, the compiler recognizes these four pairs of transition definitions and treats them as special cases. Ordinarily, the compiler generates one Java method for each IOA transition. When translating MPI transitions, the compiler generates a single method for each *pair* of transitions. The relevant appropriate MPI method invocation is sandwiched between the translations of the effects of the output and input transition in each pair.

When invoked at run time, the resulting method does all the work of the output effect, performs the MPI call, and then does the work of the input effect. Since the design uses only MPI calls that will not block, the effect as a whole will not block. (The `receive` output transition that is translated into a **receive** method invocation is enabled only after confirming, using **Iprobe**, that a message is available. Therefore, as used, the **receive** method invocation will not block.) As a result, the input half of the pair (the `resp_`* transition) does not need to be scheduled. The input half is executed automatically (without returning to the schedule loop) when the MPI call returns.

Recognizing MPI transitions To recognize these transitions, the IOA compiler uses three criteria: the names and kinds of the transitions and the pattern of the sorts of their parameters. Recognizing the sorts of the parameters is not entirely straightforward as the `Msg` and `Node` sorts are themselves type parameters to the mediator automata. Thus, the names of the sorts actually used cannot be known until the mediator automata are composed with the algorithm and buffer automata to form the node automaton. The compiler, therefore, cannot use the names of the sorts when identifying the transitions.

First, messages may be of any sort, so the `Msg` sort is treated as a "don't care". The first parameter to `Isend` or to `resp_receive` may be anything. Second, rather than require the `Node` sort to have a particular name, the compiler verifies the implementation class for the sorts in the parameter positions where the `Node` sort is expected. In particular, that implementation class must implement the Java interface **ioa.runtime.adt.MPINode**.

Manipulating parameters In the usual case, transition parameters are translated directly into method actuals in the Java code. The values of these method actuals are set at runtime by the translated **fire** statements in the schedule. For MPI transitions, this scheme works for the four output transitions (*i.e.*, `Isend`, `test`, `Isend`, and `receive`) but not for the four `resp_`* transitions.

The parameters of the output transitions contain the necessary information to generate the MPI method invocation. Not all the MPI method actual parameters are represented in the model because our implementation holds some of the parameters constant. The parameters of the four MPI input transitions are either duplicates of the output transition parameters or represent the return value of the associated MPI Java method invocation. However, the MPI return value is not available to the schedule writer.

For instance, the **Isend** method requires the message (any **java.lang.Object**) to be sent and an identifier for the destination (an **int**) as actual parameters and returns a handle (an **mpi.Request** object). In the `SendMediator` code, the actual parameters of the Java invocation of **Isend** are specified by the parameters to the `Isend` transition. The message is specified by the first parameter and the destination is specified by the third parameter to `Isend`. The return value of the Java invocation is a handle, which corresponds to the first parameter to `resp_Isend`.

To treat the return value appropriately in the generated Java code, the compiler relies on the *exact* form of those transitions. The compiler inserts the MPI call just at the place where the MPI return value is actually needed. For example, the call to **Isend** occurs in the translation of the first assignment in the effect of the `resp_Isend` transition:

```
input resp_Isend(handle, i, j)
  eff handles := handles ⊢ handle;
      status := idle.
```

A minor technical complication arises when the composer is used to prepare input for the compiler. Above, we said the parameters of the input transitions that do not represent the return value are "duplicates" of the output transition parameters. However, although the parameters correspond in the original mediator automaton code (see Figure 4.9, by the time they emerge from the composer as part of the expanded node automaton the names will likely have changed (see Appendix A.1). In Figure 4.9, the channel endpoints are named `i` and `j` in both transitions. However, in Appendix A.1 (the actual input to the compiler), `Isend` is parameterized by `N12` and `N13` while `resp_Isend` is parameterized by `N14` and `N15`. When the compiler pairs up the transitions and matches the sorts it also maps the parameter names of the input transition to those of the corresponding parameters of the output transition. (While the transition parameters do not happen to appear in the effects clauses of the LCR example, the parameters can appear as map indices in algorithms using other network topologies.)

6.4.2 Translating Buffer Transitions

The buffer input and output actions described in Section 3.4 are another special case in our translation. As described in that section, our design implements complete buffer automata like the one shown in Figure 3.4 even though the automaton produced by the interface generator and submitted to the compiler omits several actions. (Compare the previous figure to Figure 3.5). The omitted actions perform two functions. As described below, some of the actions interact with the external environment (actual I/O), and the others place invocations in or remove invocations from the `stdin` and `stdout` sequences, respectively.

Action invocation S-expressions The "missing" actions are hard coded into the generic automaton class in the standard library. Internally, input and output invocations are represented as `IOA_Invocation` tuples. Externally, invocations are represented as S-expressions that can be used to construct or can be constructed from the internal representation of `IOA_Invocation` tuples.

We require every datatype implementation class to have a static method to construct an instance of itself from an S-expression. Similarly, every implementation class must have a method to generate an S-expression representation of itself. (These requirements are enforced because every implementation class subclasses the abstract class **ioa.runtime.adt.ADT**.) Therefore, invocations of any input action of any automaton can be parsed from such S-expressions. Input to the automaton and output from the automaton are represented by standard Java file I/O. These S-expressions are then parsed and the internal representations of the appropriate `IOA_Invocation` generated.

Representing shorthand sorts (enumerations, tuples, and unions) as S-expressions is slightly tricky because, in the standard datatype library, there is only one class to implement each kind of shorthand sort. For example, all tuples are implemented by **ioa.runtime.adt.TupleSort**. The S-expressions representation of a shorthand value must include the structure of the particular shorthand sort along with the value.

For example, the input invocation `vote(1)` that signals node 1 of an LCR system to participate in an election looks like this:

```
((ioa.runtime.adt.TupleSort
  ((action (ioa.runtime.adt.EnumSort(3.0 RECEIVE SEND leader vote)))
   (params (ioa.runtime.adt.SeqSort
            ((ioa.runtime.adt.UnionSort
            ((ioa.runtime.adt.IntSort 1)
             (ioa.runtime.adt.EnumSort(0.0 Int)))))))))
```

The tuple containing that invocation has two fields: `action` and `params`. Note that the names of IOA sorts (*e.g.*, `IOA_Invocation`) do not appear in the S-expression. The external representation for IOA

datatypes references only the Java implementation classes. The automaton specific information — the name of the action and the pattern of sorts of its parameters — are buried inside the S-expression. The `action` field contains an element of an enumeration (`IOA_Action`). In this case, the enumeration of actions includes `RECEIVE`, `SEND`, `leader`, and `vote`. The particular action represented is element three of the zero-indexed list (*i.e.*, `vote`). Similarly, the parameter field consists of a sequence of one element. That element is a union (`IOA_Parameter`) containing exactly one type, an integer tagged by the label `Int`. The set of tags for the union are represented as an enumeration.

I/O threads As mentioned in Section 3.4, each node automaton is implemented by three Java threads. The *scheduled* thread executes those methods emitted by the compiler in the automaton-specific subclass. Two other threads implement input and output. The *stdin* thread parses input invocations and appends such invocations to the `stdin` sequence. The *stdout* thread removes invocations from the `stdout` sequence and outputs S-expression representations for them to the console.

The `stdin` and `stdout` sequences are, therefore, shared between the actions translated into the scheduled thread and the stdin and stdout threads. While other IOA variables are translated into single Java variables in the emitted subclass, the translation of `stdin` and `stdout` is more complicated. Since the I/O threads are implemented in the generic automaton class, they are independent of the submitted automaton. In particular, they cannot reference the automaton-specific variable containing these queues. In fact, in the automaton submitted for compilation, the I/O queues are not simple stand-alone variables. They are fields in some more complicated tuple. For example, in the `LCRNode` automaton shown in Appendix A.1, `stdin` and `stdout` are fields of the tuple variable `I`.

Therefore, the generic class includes two static variables to contain the Java representation of these sequences. It is actually these static variables that are shared between the threads. In the main thread, each access to the I/O queues is translated into a sequence of statements. Assignments to the I/O queues are converted into three assignments that read the canonical variable into the local copy, modify it locally, and then write it back to the canonical location. References to the I/O queues must copy the canonical value into the local copy before use. To prevent corruption of these static variables, the compiler uses the Java **synchronized** construct to protect queue accesses.

6.5 Translating Schedules

In our translation each IOA transition is translated into a Java method. The result of translating the NDR schedule of the IOA program is the main loop of the generated Java program. On every iteration of the schedule loop, the scheduler picks an action to fire. Translation of NDR schedule programs is nearly identical to the code generation described above for NDR **det** blocks. The chief distinction is that no infinite loop is automatically wrapped around the method that implements the schedule. When the schedule method returns, execution ends.

Instead of **yield** statements, schedule blocks contain **fire** statements. A fire statement translates into an invocation of the method generated by the translation of the **fire**d transition.

Translating det blocks for initialization The first method invoked by a schedule method is a special **initialization** method. That method is the result of translating the **det** block associated with the automaton **initially** clause, if any. Unlike choice **det** blocks, but like schedules, the **initially det** blocks are not wrapped in a loop. The block executes once and returns to the schedule. Neither **yield** nor **fire** statements appear in these NDR programs.

Chapter 7

Translation Correctness

Beware of bugs in the above code; I have only proved it correct, not tried it.

— Donald Knuth [68]

This chapter argues the correctness of our compilation method. We claim the distributed system created by compiling node automata and running the resulting Java programs linked with MPI and our datatype libraries faithfully implements the IOA system design submitted for compilation. The IOA compiler preserves the behavior at the boundary between the system and its environment as shown schematically in Figure 7.1. We assert that the externally visible behaviors the compiled system will exhibit are a subset of the externally visible behaviors permitted by the specification. We make this claim formally by proving Theorem 7.2 in Section 7.3.

For this theorem to apply to the IOA compiler it must be the case that the system submitted for compilation is structured as described in Chapter 3, the network behaves according to our model introduced in Chapter 4, the NDR annotations of the system produce valid values as discussed

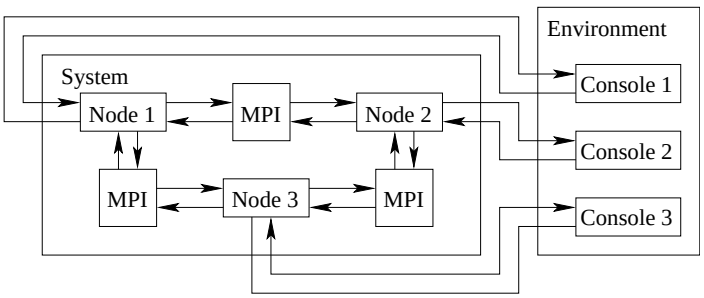

Figure 7.1: A compiled IOA system consists of a composition of node and MPI automata interacting with the environment.

in Chapter 5, and hand-coded datatype libraries correctly implement their LSL specifications as mentioned in Chapter 6.

The correctness condition for the compiler is global. We require that the external behavior is preserved for the system as a whole, not just at individual nodes. That is, we show that the multi-threaded Java programs running on multiple, concurrently-operating nodes (and not using any global synchronization) preserves the appearance of the sequential execution model of the global system I/O automaton. Our proof of correctness, however, proceeds node by node. Theorem 7.1 asserts that the externally visible behaviors a compiled node will exhibit are a subset of the externally visible behaviors permitted by its specification. We then use standard pasting lemmas to show the desired global property as a corollary.

Our approach to showing the correctness of each node is to model the compiled Java code as itself being an I/O automaton. Our model of the target code is that it is just like the source code except in that it can take (arbitrarily) small atomic steps and that these micro-steps may be interleaved across threads and nodes. We then augment this model with history variables to delay or accelerate the effects of these small steps to point in the execution where the externally visible action occurs. We define a function mapping the states of the source automaton to the states of the augmented model of the Java program. We prove the function is a refinement mapping by showing a step correspondence between our augmented model of the compiled code and the source automaton. Finally, we use this refinement mapping to conclude that the set of traces of our basic model of the Java code is a subset of the set of traces of the to the source automaton.

Separately, Section 7.4 cleans up a technical issue with the notion of input-delay insensitivity introduced in Chapter 3. In that chapter, we required that an IOA system submitted for compilation must behave correctly (as defined by the programmer) even if each node automaton is composed with a buffer automaton that delays inputs. Unfortunately, the buffer automaton we defined in that section alters the interface to the system by adding extra output actions to implement a handshake protocol. Theorem 7.7 asserts that a simpler buffer automaton that does not implement the handshake protocol is equivalent to the one given in Chapter 3. The simpler buffer automaton does not alter the interface of the submitted system.

7.1 MacroSystem

A complete distributed system suitable for compilation is described in IOA by the composition of `MPI` and the scheduled IOA programs for all the nodes. We refer to the combination of these automata as the `MacroSystem` automaton. We refer to the individual node program at each host as $\texttt{N}_i$. Each $\texttt{N}_i$ consists of more than the IOA program submitted for compilation. In Section 3.4, we distinguished between the set of actions that were conceptually part of the buffer automaton and the smaller set

```
automaton LCRSystem(MPIsize: Int)
  assumes Injective(Int, Int, Name)
  components
    N[i, size, name:Int]: LCRNode(i, size, name)
                        where 0 ≤ i ∧ i ≤ MPIsize ∧
                              size = MPIsize ∧ name = Name(i);
    M[i, j:Int]: MPI(Int, Int, i, j)
                         where 0 ≤ i ∧ i ≤ MPIsize
                                       ∧ 0 ≤ j ∧ j ≤ MPIsize
```

Figure 7.2: `LCRSystem` automaton using MPI channels. For the purposes of specifying correct behavior of the system, the `LCRSystem` automaton specifies the complete `LCRProcessInterface` buffer automaton among its components.

of actions actually emitted by the interface generator. (For example, the complete buffer automaton for LCR is shown in Figure 3.4 while the emitted version is shown in Figure 3.5.) Each $\texttt{N}_i$ node automaton in `MacroSystem` includes the complete buffer automaton among its components.

```
Injective(S, T, F): trait
  introduces
    F: S → T
  asserts with x, y: S
    F(x) = F(y) ⇒ x = y
```

Figure 7.3: LSL trait `Injective` specifying an injective function

Figure 7.2 shows the definition of the `MacroSystem` program for the LCR example. The system specification automaton assumes the existence of an injective function `Name` from node indices to node names. The LSL trait `Injective` introducing the function is shown in Figure 7.3.

Since Theorem 7.1 applies to all node automata submitted to the compiler, we now introduce a nomenclature to describe their syntactic elements. We also describe our model of NDR schedules in greater detail. We describe each of the elements of a macro-node automaton in turn below.

Action signature and threads As we alluded to in Section 3.4, we partition the actions of $\texttt{N}_i$ into three sets or *threads*. These IOA threads model the three Java threads in the node implementation. Any action that appears in the scheduled node automaton submitted for compilation is in the *main* or *schedule* thread. Those actions derived from the algorithm automaton, from the mediator automata, and from the buffer automaton as emitted by the interface generator are part of the main thread. The actions in $\texttt{N}_i$ derived from the console input actions of the algorithm automaton but omitted from the emitted buffer automaton make up the *stdin* thread. The omitted buffer actions derived from the console output actions of the algorithm automaton make up the *stdout* thread.

I/O schedules As described in Section 5.1.2, by annotating the program with an NDR schedule block, the programmer is implicitly adding a special internal schedule action to the automaton. We model the behavior of the two Java I/O threads by defining similar schedule NDR programs for the

IOA I/O threads. The output schedule is extremely simple. Whenever there is an invocation on the `stdout` queue, the schedule fires the matching output action to dequeue the invocation. The input schedule appends invocations onto the `stdin` queue whenever the `appendInvocation` method is enabled and then immediately fires the `inReady` signaling action. NDR code describing such schedules is shown in Figures 7.4 and 7.5. Note, input actions are not scheduled but are guaranteed to occur only after signaling outputs have occurred by the use of handshake protocols.

```
schedule
do
  while true do
    if valid then
       fire internal appendInvocation(MPIrank)
       fire output inReady(MPIrank)
    fi;
  od
od
```

Figure 7.4: NDR schedule block for the input thread

```
schedule
do
  while true do
    if stdout ≠ {} ∧
       (((head(stdout).action) = leader) ∧
       (len(head(stdout).params)) = 1) ∧
       (tag(head(stdout).params[0])) = Int) ∧
       (head(stdout).params[0].Int) = MPIrank
    then
       fire output leader(MPIrank)
    fi;
od
```

Figure 7.5: NDR schedule block for the output thread

States The state of each $\texttt{N}_i$ includes three special PC variables, $\texttt{PC}_{sched}$, $\texttt{PC}_{stdin}$, $\texttt{PC}_{stdout}$. We say that a PC variable *controls* actions in the thread with which it is associated. The precondition of each locally-controlled transition π requires that the controlling PC contains the identifier for π. Initially, $\texttt{PC}_{sched} = \texttt{PC}_{stdin} = \texttt{PC}_{stdout} = \texttt{schedule}$.

The variables of $\texttt{N}_i$ are also associated with threads. Except for the `stdin` and `stdout` queues, each variable of $\texttt{N}_i$ is accessed and modified only by actions in a single thread. Let $\texttt{ST}_{sched}$ be the set of variables other than $\texttt{PC}_{stdin}$ accessed by only the scheduled thread. Let $\texttt{ST}_{stdin}$ and $\texttt{ST}_{stdout}$ be defined analogously. The `stdin` and `stdout` queues are shared by the main thread and the I/O threads and are not included in $\texttt{ST}_{sched}$, $\texttt{ST}_{stdin}$, or $\texttt{ST}_{stdout}$. For notational convenience, we often treat each of the sets of variables $\texttt{ST}_{sched}$, $\texttt{ST}_{stdin}$, or $\texttt{ST}_{stdout}$ as if it were a single variable.

Transitions The precondition of each transition definition π in $\texttt{N}_i$ is a predicate $\textbf{pre}_\pi$ which, when conjoined with the action and transition **where** clauses of π, defines a set of *prestates* of π. The set of prestates of π is the subset of the states $\texttt{N}_i$ in which π is enabled. The effects of each transition definition π in $\texttt{N}_i$ defines a computable function f_π from states of $\texttt{N}_i$ to states of $\texttt{N}_i$. The image of f_π when applied to the prestates of π to is the set of *poststates* of π. In all the poststates of each action π other than the special schedule transition the value of the controlling PC is `schedule`. In the poststates of the schedule action, the value of the controlling PC is the label of some action π (including possibly `schedule`) in the controlling thread.

Figure 7.6 shows a transition equivalent to a non-schedule transition π in the schedule thread of $\texttt{N}_i$. It is worth noting that the transition function f_π is the identity with respect to variables not accessed by the controlling thread of π. If v is some variable or set of variables of $\texttt{N}_i$ and S is a state of $\texttt{N}_i$ then we use the notation $f_\pi(v)$ to mean the restriction of f_π to v. That is, $f_\pi(v)$ is shorthand for $f_\pi(S).v$.

internal $\pi(p_1 : P_1, p_2 : P_2, ...)$ **where** P
  pre $\texttt{PC}_{sched} = \pi$;
    $\textbf{pre}_\pi(\texttt{ST}_{sched}, \texttt{stdin}, \texttt{stdout})$
  eff
    ensuring
      $\texttt{ST}'_{sched} = f_\pi(\texttt{ST}_{sched}) \wedge$
      $\texttt{stdin}' = f_\pi(\texttt{stdin}) \wedge$
      $\texttt{stdout}' = f_\pi(\texttt{stdout}) \wedge$
      $\texttt{PC}'_{sched} = \texttt{schedule}$

Figure 7.6: A transition definition equivalent to the transition definition of π of the schedule thread of the macro-node automaton $\texttt{N}_i$

7.2 μSystem

We model the compiled Java code itself as an IOA program μ`System` which takes many small atomic steps that may be interleaved across threads and nodes. For each transition `MacroSystem` executes, μ`System` must execute a *sequence* of transitions to achieve the same effect. We do not specify the granularity of these micro-steps. Rather, we assert that the micro-steps are atomic with respect to thread interleaving and node concurrency. Thus, a micro-step might represent a Java statement or a machine instruction. We define each μ`System` automaton such that *if* a sequence of these micro-transitions executes without interruption or interleaving, the cumulative effect of the sequence corresponds to the effect of the corresponding transition of the `MacroSystem` automaton. In addition, we include in the definition of the derivation of the μ`System` automaton a locking discipline that

reflects our use of Java synchronized methods to control the effects of such interleaving. By stating and proving Theorem 7.1 below, we are asserting that even in the presence of interleaving the node system behaves correctly.

Like the `MacroSystem` automaton, the μ`System` automaton is composed of the `MPI` automaton and IOA automata for each node automaton in the system. However, the IOA node programs in the μ`System` are not the node programs in the `MacroSystem`. Rather, each node automaton is a component of the μ`System` automaton *derived from* a corresponding component of the `MacroSystem` automaton. This derivation is intended to correspond to the process of compiling a single node automaton. Essentially, we represent compiling a node specification as breaking up its atomic steps into many small steps that actually are atomic in the implementation. What makes this division of steps interesting is that our model of compilation also allows these steps to be interleaved across threads and to acquire and release locks.

7.2.1 Deriving a micro-node from a macro-node

We now describe the micro-step automaton $\mu\mathtt{N}_i$ that models the Java code produced by the IOA compiler to implement the macro-node automaton $\mathtt{N}_i$. Note, the IOA program μ`System` and its component node automata $\mathtt{N}_i$ are only conceptual; no such IOA programs are ever produced. The method of deriving $\mu\mathtt{N}_i$ from $\mathtt{N}_i$ described below gives a correctness condition for the relevant characteristics of the IOA compiler. That is, *if* $\mu\mathtt{N}_i$ is an accurate model of the code generated by the compiler, the compilation process preserves safety properties of the submitted automaton.

Signature For each action π of $\mathtt{N}_i$, $\mu\mathtt{N}_i$ has a *sequence* of actions $\pi_1, \pi_2, \ldots, \pi_x, \ldots \pi_z$. If π is an output action, then π_z is an output action. If π is an input action, then π_1 is an input action. All other micro-actions are internal. Thus, the external signature of $\mathtt{N}_i$ and $\mu\mathtt{N}_i$ are identical. We say that π is also the name of such a micro-action sequence.

Three technical points are worth noting. First, we assume that every micro-action sequence contains at least two actions. Second, for notational simplicity, we use z as the index of the last micro-action of every micro-sequence. Technically, z should be a function $z(\pi)$ and not a constant. (Alternatively, we could pad every sequence with no-op micro-actions so that every sequence has the same length.) Third, the first micro-action of an input sequence and the last micro-action of an output sequence must be named π rather than π_1 or π_z in order for the external interface of $\mu\mathtt{N}_i$ to match that of $\mathtt{N}_i$. However, for consistency and simplicity in the discussions below, we always refer to these actions using the subscripted form. We finesse this issue by defining the trace of such actions to be π.

States For every state variable of $\mathtt{N}_i$, $\mu\mathtt{N}_i$ has a directly corresponding state variable. We define the variable sets $\mu\mathtt{ST}_{sched}$, $\mu\mathtt{ST}_{stdin}$, and $\mu\mathtt{ST}_{stdout}$ analogously to $\mathtt{ST}_{sched}$, $\mathtt{ST}_{stdin}$, and $\mathtt{ST}_{stdout}$. In the micro-step node automaton each micro-program counter ($\mu\mathtt{PC}_{sched}$, $\mu\mathtt{PC}_{stdin}$, and $\mu\mathtt{PC}_{stdout}$) is a pair. The first element of each $\mu\mathtt{PC}$ is the name of the micro-action sequence being executed in its thread. The second element of the pair is an index denoting which micro-action in the sequence is being executed. Finally, each $\mu\mathtt{N}_i$ has two special lock variables $\mathtt{lock}_{stdin}$ and $\mathtt{lock}_{stdout}$, which we will discuss below. The lock variables are shared between threads and, therefore, do not appear in the $\mu\mathtt{ST}$ sets.

Transitions For each transition definition π with precondition $\mathbf{pre}_\pi$ and effect function f_π of $\mathtt{N}_i$, $\mu\mathtt{N}_i$ has a sequence of transition definitions $\pi_1, \pi_2, \ldots, \pi_x, \ldots, \pi_z$. The precondition of each locally-controlled action π_x is a predicate $\mathbf{pre}_{\pi,x}$. If π is locally-controlled, then the predicate $\mathbf{pre}_{\pi,1}$ is the same as $\mathbf{pre}_\pi$ except that where $\mathbf{pre}_\pi$ requires the controlling $\mathtt{PC}$ to name π, $\mathbf{pre}_{\pi,1}$ requires the controlling $\mu\mathtt{PC}$ to be the pair $[\pi, 1]$. The precondition $\mathbf{pre}_{\pi,x}$ of each other locally-controlled micro-action requires only that the controlling $\mu\mathtt{PC}$ be the pair $[\pi, x]$. The effects of each micro-action π_x defines an effects functions $f_{\pi,x}$.

Let f^*_π be the composition of the micro-effects functions $f_{\pi,1}, f_{\pi,2}, \ldots, f_{\pi,x}, \ldots, f_{\pi,z}$. Let S be a state in which π is enabled in the macro-node automaton $\mathtt{N}_i$ and S' be state that results from executing π in S. Let s be a state of the micro-node automaton $\mu\mathtt{N}_i$ in which π_1 is enabled and s' be a state that results from executing the sequence of transitions $\pi_1, \pi_2, \ldots \pi_z$ in s. Let $P(s)$ be the projection of s onto S. We require that $f_\pi(S) = P(f^*_\pi(s))$. That is, if

- two corresponding automata $\mathtt{N}_i$ and $\mu\mathtt{N}_i$ begin in corresponding states S and s,
- the execution of transition π in state S results in state $S' = f_\pi(S)$, and
- the *uninterrupted* execution of the sequence of transitions $\pi_1, \pi_2, \ldots, \pi_z$ in state s results in state $s' = f^*_\pi(s)$,

then the post-state s' of the last micro-action in the sequence must correspond to the post-state S' of the macro-action.

In addition to whatever other work it performs, each micro-action — other than the last — in a sequence increments the index element of its controlling $\mu\mathtt{PC}$. The last micro-action in each non-schedule sequence assigns the first element of its controlling $\mu\mathtt{PC}$ to $\mathtt{schedule}$ and sets the index element to one. Similarly, executing a micro-action sequence for a schedule action in a micro-state s must result in a state where the first element in the controlling $\mu\mathtt{PC}$ equals the $\mathtt{PC}$ that results from applying the macro-schedule action in $S = P(s)$.

Note, while $\mu\mathtt{N}_i$ must step sequentially through a sequence of micro-actions in one thread, legal executions of $\mu\mathtt{N}_i$ include those in which the micro-actions of one sequence are interleaved with

those of sequences in different threads (either at the same node or at others) or with steps of the `MPI` automata. Theorem 7.2, below, argues such interleavings do not alter the externally visible behavior of the whole system.

Locks We model the synchronized methods described in Section 6.4.2 using two special lock variables, one each for the shared I/O queues `stdin` and `stdout`. In the micro-action sequences derived from macro-actions that are translated to synchronized blocks (*i.e.*, actions that reference `stdin` or `stdout`), we include a special micro-step to grab a lock on the appropriate queue. In general we do not specify the granularity of the micro-steps or the micro-effects functions $f_{\pi,i}$. However, we do require that there are special micro-steps to grab and release locks. That is, locking must be atomic with respect to thread interleaving. In any such micro-action sequence π, the first micro-action in the sequence π_1 grabs the lock and the last micro-step π_z releases it. If another action already has the lock, the micro-action is a no-op. The controlling μ`PC` remains $[\pi, 1]$, in effect spinning on the lock. Note, this spinning blocks only the thread attempting to grab the lock.

Since only internal actions ever grab locks, spinning does not change the trace of the automaton. In the specification of the buffer automata in Section 3.4, we separate the `appendInvocation` method from the console input specifically to make such spinning on a lock internal to the buffer automaton. Since locks are state variables of the $\mu\mathtt{N}_i$ node automata, locks are local, not global.

7.3 Compilation Correctness Theorems

Our proof of correctness for the compiler consists of two theorems: one for compilation of a single node in the system and one for the whole system. Due to the requirement that systems be described in node-channel form and our assumptions about the correctness of our channel model, proving global correctness is simple once node-by-node correctness has been shown. In Sections 7.3.1 and 7.3.2, we present Theorems 7.1 and 7.2 and show how the latter follows immediately from the former. In Section 7.3.3, we discuss history variables and their definition in IOA. In Section 7.3.4, we define an augmented micro-action node automaton $\mu\hat{\mathtt{N}}_i$ by adding history variables to $\mu\mathtt{N}_i$. In the Section 7.3.5, we prove a series of invariants about this augmented automaton. In Section 7.3.6 we demonstrate a refinement mapping from the augmented micro-action automaton to the macro-action automaton using the invariants. The proof of Theorem 7.1 concludes in Section 7.3.6 by showing a forward simulation from $\mu\mathtt{N}_i$ to $\mathtt{N}_i$.

7.3.1 Node Correctness Theorem

The theorem for correctness of the node-by node compiler says that the externally visible behaviors of $\mu\mathtt{N}_i$ are a subset of those of $\mathtt{N}_i$. Thus, if $\mu\mathtt{N}_i$ correctly models the generated Java code, the

NDR annotations of the node produce valid values, and the hand-coded datatype libraries correctly implement their LSL specifications, then the compiled node will exhibit only behaviors specified by the system designer in $\texttt{N}_i$.

Theorem 7.1 (Node Correctness) The set of traces of $\mu\texttt{N}_i$ is a subset of the set of traces of $\texttt{N}_i$.

We delay the proof of Theorem 7.1 until Section 7.3.6 where we demonstrate a refinement mapping from the augmented micro-action automaton $\mu\hat{\texttt{N}}_i$ to the macro-node automaton $\texttt{N}_i$.

7.3.2 Global System Correctness Theorem

The global correctness theorem is a corollary of Theorem 7.1. The theorem for correctness of the entire compiled system says that the externally visible behaviors of μ`System` are a subset of those of `MacroSystem`. Thus, if the system submitted for compilation is correctly structured, the MPI system behaves according to our `MPI` model, the NDR annotations of the system produce valid values, the hand-coded datatype libraries correctly implement their LSL specifications, and $\mu\texttt{N}_i$ correctly models the generated Java code, then the compiled system will exhibit only behaviors specified by the system designer in `MacroSystem`.

Theorem 7.2 (Global Correctness) The set of traces of μ`System` is a subset of the set of traces of `MacroSystem`.

Proof: Theorem 7.2 follows immediately from Theorem 7.1 and Theorem 8.10 of [81]. The latter "pasting theorem" implies that if the set of traces of each component A_i of one composite automaton A is a subset of the set of traces of corresponding component B_i of another composite automaton B, then the set of traces of A is a subset of the set of traces of B. ■

7.3.3 History variables

Informally, history variables are augmentations to the state of an automaton that are used to record previous states or actions of the automaton or functions of those values. Formally, history variables can be defined in terms of history relations. (See Section 2.1.2.) Consider a variable v, a base automaton B and the augmented automaton A formed by adding v (and references to and manipulations of v) to B. The variable v is a history variable of an augmented automaton whenever there is a history relation from B to A [83].

In IOA, the following syntactic conditions are sufficient (and much stronger than necessary) to ensure variables $h_1, h_2, \ldots$ are history variables of an IOA automaton A. Given a base automaton B with state variables $v_1, v_2, \ldots$ there is a history relation from B to augmented automaton A whenever the text of A is related to that of B as follows.

- A and B have the same signature.
- The state variables of A are $v_1, v_2, \ldots, h_1, h_2, \ldots$.
- A and B have the same initial value terms for $v_1, v_2, \ldots$.
- A and B have the same **initially** predicate.
- A and B have the same set of transition definitions, each with the same parameters, local variables, precondition, **where** clause, and **ensuring** predicate.
- Each transition definition of A includes the entire text of the corresponding transition of B.
- In any additional text of a transition definition, no v_i appears on the left hand side of an assignment.

To prove these syntactic restrictions are sufficient for $h_1, h_2, \ldots$ are history variables, it is necessary to show that there is always a forward simulation from B to A whose inverse is a refinement mapping from A to B. The proof is left as an exercise for the reader.

7.3.4 $\mu\hat{\mathtt{N}}_i$

To help establish the trace inclusion from $\mu\mathtt{N}_i$ to $\mathtt{N}_i$, we define automaton $\mu\hat{\mathtt{N}}_i$ by augmenting the state of micro-node program $\mu\mathtt{N}_i$ with history variables. In particular, every variable $\mathtt{v}$ in both the state of $\mu\mathtt{N}_i$ and of $\mathtt{N}_i$ (*i.e.*, all variables except the three $\mu\mathtt{PC}$ s and the two locks) is mirrored by a history variable $\overline{\mathtt{v}}$ of the same type as $\mathtt{v}$. The value of $\overline{\mathtt{v}}$ is initialized to the initial value $\mathtt{v}$. In addition, we augment $\mu\mathtt{N}_i$ with a three history PC variables, $\overline{\mathtt{PC}}_{sched}$, $\overline{\mathtt{PC}}_{stdin}$, and $\overline{\mathtt{PC}}_{stdout}$ of the same types as and initialized to the same values as $\mathtt{PC}_{sched}$, $\mathtt{PC}_{stdin}$, and $\mathtt{PC}_{stdout}$, respectively.

The history variables are updated only in the last step of a locally-controlled action sequence or the first step of an input micro-action sequence. In the last micro-action π_z of a locally-controlled sequence all history variables are assigned the values the corresponding state variables have in the poststate of π_z. Thus, the history variables mirror the state of $\mu\hat{\mathtt{N}}_i$ after the micro-action. In the first micro-action π_1 of an input sequence the values of all history variables are updated by assigning them the values that result from applying f^*_π to the values of the corresponding state variables in the prestate of π_1. That is, all history variables are assigned the values the regular variables *will* have at the end of the input micro-action sequence.

In the steps where history state variables are updated, the history PC variables also updated. In the last step of a locally-controlled micro-action sequence, the controlling $\overline{\mu\mathtt{PC}}$ is assigned the value the first element of the controlling $\mu\mathtt{PC}$ pair has in the poststate of π_z. In the first micro-action π_1 of an input action sequence, the controlling $\overline{\mu\mathtt{PC}}$ is assigned the first element of the pair that results from applying f^*_π to the value of the controlling $\mu\mathtt{PC}$ in the prestate of π_1. That is, the controlling

internal $\pi_1(q_1 : Q_1, q_2 : Q_2, ...)$ **where** Q

pre $\mu\texttt{PC}_{sched} = [\pi, 1]$;

pre$_{\pi,1}(\mu\texttt{ST}_{sched}, \texttt{stdin}, \texttt{stdout})$

eff

ensuring

if $\texttt{lock}_{stdin} = \texttt{idle}$ **then**

$\texttt{lock}'_{stdin} = \texttt{stdin} \wedge$

$\mu\texttt{ST}'_{sched} = f_{\pi,1}(\mu\texttt{ST}_{sched}) \wedge$

$\texttt{stdin}' = f_{\pi,1}(\texttt{stdin}) \wedge$

$\mu\texttt{PC}_{sched} = [\pi, 2]$

else $\mu\texttt{PC}_{sched} = [\pi, 1]$

fi

...

output $\pi_z(r_1 : R_1, r_2 : R_2, ...)$ **where** R

pre $\mu\texttt{PC}_{sched} = [\pi, z]$

eff ...;

$\overline{\mu\texttt{ST}}'_{sched} := \mu\texttt{ST}_{sched}$; $\overline{\texttt{stdin}}' := \texttt{stdin}$; $\overline{\texttt{PC}}'_{sched} := \texttt{schedule}$

ensuring

$\texttt{lock}'_{stdin} = idle \wedge$

$\mu\texttt{ST}'_{sched} = f_{\pi,z}(\mu\texttt{ST}_{sched}) \wedge$

$\texttt{stdin}' = f_{\pi,z}(\texttt{stdin}) \wedge$

$\mu\texttt{PC}'_{sched} = [\texttt{schedule}, 1]$

Figure 7.7: Sequence of transitions of the augmented micro-node automaton $\mu\hat{\texttt{N}}_i$ corresponding to a transition of the macro-node automaton $\texttt{N}_i$. The sequence corresponds to an internal transition π in the schedule thread that references the shared buffer `stdin`.

input $\phi_1(q_1 : Q_1, q_2 : Q_2, ...)$ **where** Q

eff ...;

$\overline{\mu\texttt{ST}}'_{stdin} := f^*_\phi(\mu\texttt{ST}_{stdin})$; $\quad\quad$ $\overline{\texttt{PC}}'_{stdin} := \texttt{schedule}$

ensuring

$\mu\texttt{ST}'_{stdin} = f_{\pi,1}(\mu\texttt{ST}_{sched}) \wedge$

$\mu\texttt{PC}'_{stdin} = [\phi, 2]$

...

internal $\phi_z(r_1 : R_1, r_2 : R_2, ...)$ **where** R

pre $\mu\texttt{PC}_{stdin} = [\phi, z]$

eff ...

ensuring

$\mu\texttt{ST}'_{stdin} = f_{\pi,z}(\mu\texttt{ST}_{stdin}) \wedge$

$\mu\texttt{PC}'_{stdin} = [\texttt{schedule}, 1]$;

Figure 7.8: Sequence of transitions of the augmented micro-node automaton $\mu\hat{\texttt{N}}_i$ corresponding to an input transition of the macro-node automaton $\texttt{N}_i$. The sequence corresponds to an input transition ϕ of the stdin thread.

$\overline{\mu\texttt{PC}}$ is assigned the first element of the value the controlling $\mu\texttt{PC}$ pair *will* have when the sequence completes.

The invariants we prove in the next section formally justify our claim that assigning history variables the result of applying f^*_π to the micro-action prestate correctly predicts the future values of variables. Informally, three factors make it possible to predict the value variables will have at the end of the sequence. First, all input actions (both the MPI `resp_`* actions and the buffer input actions) are deterministic. Second, input actions do not reference any variables referenced by actions in any other thread. Third, by construction, micro-action sequences in the same thread are never interleaved. (Such interleaving is preventing by checking the value of the controlling $\mu\texttt{PC}$ in the precondition of each micro-action.) The only way one sequence can affect the outcome of another is by changing the value of some variable referenced by the other sequence. The only variables shared across threads are locks and the special `stdin` and `stdout` sequences. In our translation, no input actions reference these shared variables. *Therefore, input actions are deterministic even if micro-action sequences are interleaved across threads or across nodes.*

Figures 7.7 and 7.8 show two sequences of micro-actions. The first sequence corresponds to an internal macro-transition π in the schedule thread that accesses the shared variable `stdin`. The second sequence corresponds to an input macro-transition ϕ in the stdin thread. The first sequence demonstrates our model of spin-locks and how history variables are updated in the last step of locally-controlled actions. In contrast, the result of applying the composite function f^*_ϕ to $\mu\texttt{ST}_{stdin}$ (and, thus, the result of executing the entire sequence) is assigned to $\overline{\mu\texttt{ST}}_{stdin}$ in the effects of the

first micro-action of the second sequence. The history $\overline{\mu\texttt{PC}}$ variable is also updated in the first micro-action of the second sequence.

7.3.5 Invariants

The following seven invariants of $\mu\hat{\texttt{N}}_i$ help to establish a refinement mapping from $\mu\hat{\texttt{N}}_i$ to $\texttt{N}_i$. The first invariant defines the locking discipline of the micro-node automaton. It asserts that there is no reachable state of the micro-node automaton where two actions are simultaneously in the critical-section and that locks always name the thread currently in the critical section (if any). The next five are "progress" invariants that relate the values of the five history variables to the partial progress sequences of micro-actions make in updating the five parts of the state of the micro-node automaton. The last invariant asserts that preconditions are stable over the course of a sequence of micro-actions. That is, the precondition of the macro-action remains satisfied by history variables in every step of the micro-action sequence.

The Locking Invariant says that no action sequence that accesses a shared queue can begin or proceed without the lock on that queue. Furthermore, access to the queue is exclusive to one thread. While micro-action sequences that access the shared queue can be always be scheduled, such sequences never proceed past the first micro-step without obtaining the lock. (Recall that the first micro-step of such sequences is defined to be a no-op if the thread does not have the lock. See Figure 7.7.) We do not prove this invariant. Rather, we take it as our axiomatic definition of locking.

Invariant 7.1 (Locking) In all reachable states of $\mu\hat{\texttt{N}}_i$, for all locally-controlled action sequences π in the stdin thread and ϕ in the main thread that (both) access `stdin`,

1. $\mu\texttt{PC}_{stdin} = [\pi, x] \ \wedge \ \mu\texttt{PC}_{sched} = [\phi, y] \Rightarrow x = 1 \ \vee \ y = 1$,
2. For all $x, 1 < x \leq z, \mu\texttt{PC}_{stdin} = [\pi, x] \Rightarrow \texttt{lock}_{stdin} = \texttt{stdin}$, and
3. For all $y, 1 < y \leq z, \mu\texttt{PC}_{sched} = [\phi, y] \Rightarrow \texttt{lock}_{stdin} = \texttt{schedule}$.

Likewise, in all reachable states of $\mu\hat{\texttt{N}}_i$, for all locally-controlled action sequences π in the stdout thread and ϕ in the main thread that (both) access `stdout`,

4. $\mu\texttt{PC}_{stdout} = [\pi, x] \wedge \mu\texttt{PC}_{sched} = [\phi, y] \Rightarrow x = 1 \ \vee \ y = 1$,
5. For all $x, 1 < x \leq z, \mu\texttt{PC}_{stdout} = [\pi, x] \Rightarrow \texttt{lock}_{stdout} = \texttt{stdout}$, and
6. For all $y, 1 < y \leq z, \mu\texttt{PC}_{sched} = [\phi, y] \Rightarrow \texttt{lock}_{stdout} = \texttt{schedule}$.

The $\mu\texttt{ST}_{stdin}$ Progress Invariant says that in the states where a micro-action sequence in the stdin thread has been scheduled but not begun executing, the $\mu\texttt{ST}_{stdin}$ state variable equals its corresponding history variable $\overline{\mu\texttt{ST}}_{stdin}$. Note that there may be many such states because the

micro-actions of the other two threads may be interleaved with those of the stdin thread. The invariant also states that each micro-action in the sequence affects the value of $\mu\mathtt{ST}_{stdin}$ as if it had been executed immediately after the preceding action in the sequence — without regard to any interleaved micro-actions from the other threads. Thus, value of $\mu\mathtt{ST}_{stdin}$ can always be related to the value of $\overline{\mu\mathtt{ST}}_{stdin}$ simply by knowing which micro-action sequence is executing and the index of the particular micro-action scheduled to run next.

The proof proceeds by considering the effects of micro-actions from other threads and then by six cases for which micro-action scheduled in the stdin thread. The six cases are the combinations of the kind of the micro-action sequence (input or locally-controlled) and the position of the micro-action in the sequence (first, middle, or last).

The $\mu\mathtt{ST}_{stdout}$ Progress and $\mu\mathtt{ST}_{sched}$ Progress Invariants below are completely analogous.

Invariant 7.2 ($\mu\mathtt{ST}_{stdin}$ Progress) In all reachable states of $\mathtt{N}_i$

1. For all action sequences π of $\mu\hat{\mathtt{N}}_i$,

$$\mu\mathtt{PC}_{stdin} = [\pi, 1] \Rightarrow \mu\mathtt{ST}_{stdin} = \overline{\mu\mathtt{ST}}_{stdin} \ \wedge \ \overline{\mathtt{PC}}_{stdin} = \pi,$$

2. For all input action sequence π of $\mu\hat{\mathtt{N}}_i$ and $1 < x \leq z$,

$$\mu\mathtt{PC}_{stdin} = [\pi, x] \Rightarrow f_{\pi,z}(f_{\pi,z-1}(\ldots(f_{\pi,x}(\mu\mathtt{ST}_{stdin})))) = \overline{\mu\mathtt{ST}}_{stdin}, \text{ and}$$

3. For all locally-controlled action sequences π of $\mu\hat{\mathtt{N}}_i$ and $1 < x \leq z$,

$$\mu\mathtt{PC}_{stdin} = [\pi, x] \Rightarrow f_{\pi,x}(f_{\pi,x-1}(\ldots(f_{\pi,1}(\overline{\mu\mathtt{ST}}_{stdin})))) = \mu\mathtt{ST}_{stdin}.$$

Proof: We show Invariant 7.2 by induction on the length of an execution of $\mu\hat{\mathtt{N}}_i$. In the initial state s_0 of $\mu\hat{\mathtt{N}}_i$, $s_0.\mu\mathtt{PC}_{stdin} = [\mathtt{schedule}, 1]$, $s_i.\overline{\mathtt{PC}}_{stdin} = \mathtt{schedule}$, and $s_0.\mu\mathtt{ST}_{stdin} = s_0.\overline{\mu\mathtt{ST}}_{stdin}$ by construction, so implication 7.2.1 holds and the other two are trivial.

For the inductive case, we assume the invariant holds in some reachable state s and we show that every micro-action π_x resulting in post-state s' when executed in s maintains the invariant. We proceed by case analysis on the micro-action π_x and the sequence π of which it is a part.

- If π is not in the stdin thread, then by construction $f_{\pi,x}$ is the identity on $\mu\mathtt{ST}_{stdin}$ and so $s'.\mu\mathtt{ST}_{stdin} = s.\mu\mathtt{ST}_{stdin}$ and $s'.\overline{\mu\mathtt{ST}}_{stdin} = s.\overline{\mu\mathtt{ST}}_{stdin}$. Furthermore $s'.\mu\mathtt{PC}_{stdin} = s.\mu\mathtt{PC}_{stdin}$ and $s'.\overline{\mathtt{PC}}_{stdin} = s.\overline{\mathtt{PC}}_{stdin}$. Therefore, all the expressions appearing in the invariant are unchanged from s to s' and the invariant is maintained.

- If π_x is the first micro-action in a locally-controlled sequence in the stdin thread (*i.e.*, $x = 1$), then implication 7.2.1 is non-trivial in s and, thus, $\mu\mathtt{ST}_{stdin} = \overline{\mu\mathtt{ST}}_{stdin}$. The effect of the action is to apply $f_{\pi,1}$ to s and to increment $\mu\mathtt{PC}_{stdin}$ so that $s'.\mu\mathtt{PC}_{stdin} = [\pi, 2]$. Since history variables are unchanged by the action, $s'.\overline{\mu\mathtt{ST}}_{stdin} = s.\overline{\mu\mathtt{ST}}_{stdin}$. Therefore, $s'.\mu\mathtt{ST}_{stdin} = f_{\pi,1}(s).\mu\mathtt{ST}_{stdin} = f_{\pi,1}(s).\overline{\mu\mathtt{ST}}_{stdin} = f_{\pi,1}(s').\overline{\mu\mathtt{ST}}_{stdin}$. Thus, the consequent of the implication 7.2.3 is satisfied in s'. The other two implications are trivially true in s'.

- If π is a locally-controlled micro-action sequence in the stdin thread and π_x is neither the first nor the last action in the sequence (*i.e.*, $1 < x < z$), then only implication 7.2.3 is non-trivial in s. The effect of the action applies $f_{\pi,x}$ to s. Therefore, $s'.\mu\mathtt{ST}_{stdin} = f_{\pi,x}(s).\mu\mathtt{ST}_{stdin}$ and $s'.\mu\mathtt{PC}_{stdin} = [\pi, x+1]$. Implication 7.2.1 is trivially true in s' because $x > 1$. Implication 7.2.2 is trivially true in s' because π is not an input sequence. Implication 7.2.3 remains true in s' because $f_{\pi,x}$ has been applied to both sides of the equality.

- If π_x is the last micro-action in a locally-controlled sequence in the stdin thread (*i.e.*, $x = z$), then $s'.\mu\mathtt{PC}_{stdin} = [\phi, 1]$ where ϕ is some action in the stdin thread. By construction, $s'.\mu\mathtt{ST}_{stdin} = s'.\overline{\mu\mathtt{ST}}_{stdin}$ and $s'.\overline{\mathtt{PC}}_{stdin} = \phi$ making implication 7.2.1 true in s'. Both implications 7.2.2 and 7.2.3 are trivially true in s'.

- If π_x is the first micro-action in an input sequence in the stdin thread (*i.e.*, $x = 1$), then only implication 7.2.1 is non-trivial in s and, thus, $s.\mu\mathtt{ST}_{stdin} = s.\overline{\mu\mathtt{ST}}_{stdin}$. The effect of the action applies $f_{\pi,1}$ to s and to increment $\mu\mathtt{PC}_{stdin}$ so that $s'.\mu\mathtt{PC}_{stdin} = [\pi, 2]$. By construction, $s'.\overline{\mu\mathtt{ST}}_{stdin} = f^*_\pi(s).\overline{\mu\mathtt{ST}}_{stdin}$. Therefore $s'.\mu\mathtt{ST}_{stdin} = f_{\pi,1}(s).\mu\mathtt{ST}_{stdin}$ and $s'.\overline{\mu\mathtt{ST}}_{stdin} = f_{\pi,z}(f_{\pi,z-1}(\ldots(f_{\pi,2}(s'))))).\mu\mathtt{ST}_{stdin}$. Thus, the consequent of implication 7.2.2 is satisfied in s'. Implications 7.2.1 and 7.2.3 are trivially true in s'.

- If π is an input micro-action sequence in the stdin thread and π_x is neither the first nor the last action in the sequence (*i.e.*, $1 < x < z$), then only implication 7.2.2 is non-trivial in s. The effect of the action applies $f_{\pi,x}$ to s. Therefore, $s'.\mu\mathtt{ST}_{stdin} = f_{\pi,x}(s).\mu\mathtt{ST}_{stdin}$ and $s'.\mu\mathtt{PC}_{stdin} = [\pi, x+1]$. By construction, $s'.\overline{\mu\mathtt{ST}}_{stdin} = s.\overline{\mu\mathtt{ST}}_{stdin}$. Implication 7.2.1 is trivially true in s' because $x + 1 > 1$. Implication 7.2.3 is trivially true in s' because π is not a locally-controlled sequence. Implication 7.2.2 remains true in s' because $f_{\pi,x}$ has been applied to $\mu\mathtt{ST}_{stdin}$ while the sequence of functions is reduced by the application of the same function. Therefore, both sides of the equality remain unchanged.

- If π_x is the last micro-action in an input sequence in the stdin thread (*i.e.*, $x = z$), then only implication 7.2.2 is non-trivial in s. By construction, $s'.\mu\mathtt{PC}_{stdin} = [\mathtt{schedule}, 1]$. Since $f_{\pi,z}$ is the only function in the sequence in that implication and because the action applies

$f_{\pi,z}$ to s, $s'.\mathtt{ST}_{stdin} = s'.\overline{\mu\mathtt{ST}}_{stdin}$ and, therefore, implication 7.2.1 is true in s'. Implications 7.2.2 and 7.2.3 are trivially true in s'.

■

Invariant 7.3 ($\mu\mathtt{ST}_{stdout}$ Progress) In all reachable states of $\mathtt{N}_i$, for all action sequences π of $\mu\hat{\mathtt{N}}_i$, In all reachable states of $\mathtt{N}_i$

1. For all action sequences π of $\mu\hat{\mathtt{N}}_i$,

$$\mu\mathtt{PC}_{stdout} = [\pi, 1] \Rightarrow \mu\mathtt{ST}_{stdout} = \overline{\mu\mathtt{ST}}_{stdout},$$

2. For all input action sequence π of $\mu\hat{\mathtt{N}}_i$ and $1 < x \leq z$,

$$\mu\mathtt{PC}_{stdout} = [\pi, x] \Rightarrow f_{\pi,z}(f_{\pi,z-1}(\ldots(f_{\pi,x}(\mu\mathtt{ST}_{stdout})))) = \overline{\mu\mathtt{ST}}_{stdout}, \text{ and}$$

3. For all locally-controlled action sequences π of $\mu\hat{\mathtt{N}}_i$ and $1 < x \leq z$,

$$\mu\mathtt{PC}_{stdout} = [\pi, x] \Rightarrow f_{\pi,x}(f_{\pi,x-1}(\ldots(f_{\pi,1}(\overline{\mu\mathtt{ST}}_{stdout})))) = \mu\mathtt{ST}_{stdout}.$$

Proof: The proof of Invariant 7.3 is analogous to that of Invariant 7.2. ■

Invariant 7.4 ($\mu\mathtt{ST}_{sched}$ Progress) In all reachable states of $\mathtt{N}_i$

1. For all action sequences π of $\mu\hat{\mathtt{N}}_i$,

$$\mu\mathtt{PC}_{sched} = [\pi, 1] \Rightarrow \mu\mathtt{ST}_{sched} = \overline{\mu\mathtt{ST}}_{sched},$$

2. For all input action sequence π of $\mu\hat{\mathtt{N}}_i$ and $1 < x \leq z$,

$$\mu\mathtt{PC}_{sched} = [\pi, x] \Rightarrow f_{\pi,z}(f_{\pi,z-1}(\ldots(f_{\pi,x}(\mu\mathtt{ST}_{sched})))) = \overline{\mu\mathtt{ST}}_{sched}, \text{ and}$$

3. For all locally-controlled action sequences π of $\mu\hat{\mathtt{N}}_i$ and $1 < x \leq z$,

$$\mu\mathtt{PC}_{sched} = [\pi, x] \Rightarrow f_{\pi,x}(f_{\pi,x-1}(\ldots(f_{\pi,1}(\overline{\mu\mathtt{ST}}_{sched})))) = \mu\mathtt{ST}_{sched}.$$

Proof: The proof of Invariant 7.4 is analogous to that of Invariant 7.2. ■

The `stdin` Progress Invariant says that in any state where neither the stdin thread nor the schedule thread has the lock on the shared `stdin` queue, the `stdin` queue is equal to its corresponding

history variable $\overline{\texttt{stdin}}$. Second, the invariant says that in the states where a micro-action sequence that accesses the `stdin` queue has been scheduled but not begun executing in one thread, either the other thread has the lock on the queue, or the queue equals its history variable. Third, the invariant says that each micro-action in the sequence affects the value of `stdin` as if it had been executed immediately after the preceding action in the sequence — without regard to any interleaved micro-actions from the other threads. Thus, the value of `stdin` can always be related to the value of $\overline{\texttt{stdin}}$ simply by knowing which micro-action sequence is executing and the index of the particular micro-action scheduled to run next.

The proof proceeds by considering the effects of micro-actions from the stdout thread, micro-actions that do not access the shared variable, and then micro-actions that do access the shared variable. Recall, that no input actions access the shared queue. For the locally-controlled actions of the stdin thread that access the shared queue, we consider three cases depending on the position of the micro-action in the sequence (first, middle, or last). We apply Locking Invariant 7.1 to exclude interleavings where micro-actions from different threads both access the shared queue.

The three cases for locally-controlled actions of the schedule thread are analogous.

The `stdout` Progress Invariant below is completely analogous.

Invariant 7.5 (`stdin` Progress) In all reachable states of $\texttt{N}_i$

1. $\texttt{lock}_{stdin} = \texttt{idle} \Rightarrow \texttt{stdin} = \overline{\texttt{stdin}}$,

2. For all locally-controlled action sequences π in the stdin thread of $\mu\hat{\texttt{N}}_i$ that reference `stdin`,

$$\mu\texttt{PC}_{stdin} = [\pi, 1] \Rightarrow (\texttt{stdin} = \overline{\texttt{stdin}} \ \vee \ \texttt{lock}_{stdin} = \texttt{schedule}),$$

3. For all locally-controlled action sequences π in the stdin thread of $\mu\hat{\texttt{N}}_i$ that reference `stdin` and $1 < x \leq z$,

$$\mu\texttt{PC}_{stdin} = [\pi, x] \Rightarrow f_{\pi,x}(f_{\pi,x-1}(\ldots(f_{\pi,1}(\overline{\texttt{stdin}})))) = \texttt{stdin},$$

4. For all locally-controlled action sequences ϕ in the schedule thread of $\mu\hat{\texttt{N}}_i$ that reference `stdin`,

$$\mu\texttt{PC}_{sched} = [\phi, 1] \Rightarrow (\texttt{stdin} = \overline{\texttt{stdin}} \ \vee \ \texttt{lock}_{stdin} = \texttt{stdin}), \text{ and}$$

5. For all locally-controlled action sequences ϕ in the schedule thread of $\mu\hat{\texttt{N}}_i$ that reference `stdin` and $1 < y \leq z$,

$$\mu\texttt{PC}_{sched} = [\phi, y] \Rightarrow f_{\phi,y}(f_{\phi,y-1}(\ldots(f_{\phi,1}(\overline{\texttt{stdin}})))) = \texttt{stdin}.$$

Proof: We show Invariant 7.5 by induction on the length of an execution of $\mu\hat{\mathtt{N}}_i$. In the initial state s_0 of $\mu\hat{\mathtt{N}}_i$, $\mu\mathtt{PC}_{sched} = \mu\mathtt{PC}_{stdin} = [\mathtt{schedule}, 1]$, $\mathtt{lock}_{stdin} = \mathtt{idle}$, and $\mathtt{stdin} = \overline{\mathtt{stdin}}$ by construction. In s_0, implication 7.5.1 is true, implications 7.5.3 and 7.5.5 are trivial, and implications 7.5.2 and 7.5.4 hold because the first conjunct of each consequent is true.

For the inductive case, we assume the invariant holds in some reachable state s and we show that every micro-action π_x resulting in post-state s' when executed in s maintains the invariant. We proceed by case analysis on the micro-action π_x and the sequence π of which it is a part.

- If micro-action sequence π is in the `stdout` thread then, by construction, $f_{\pi,x}$ is the identity on `stdin` and $\overline{\mathtt{stdin}}$, so $s'.\mathtt{stdin} = s.\mathtt{stdin}$ and $s'.\overline{\mathtt{stdin}} = s.\overline{\mathtt{stdin}}$. Furthermore, $s'.\mu\mathtt{PC}_{stdin} = s.\mu\mathtt{PC}_{stdin}$, $s'.\mu\mathtt{PC}_{sched} = s.\mu\mathtt{PC}_{sched}$, and $s'.\mathtt{lock}_{stdin} = s.\mathtt{lock}_{stdin}$. Therefore, all the expressions appearing in the invariant are unchanged from s to s' and the invariant is maintained.

- If micro-action sequence π does not reference `stdin` then $f_{\pi,x}$ is the identity on `stdin` and $\overline{\mathtt{stdin}}$ so $s'.\mathtt{stdin} = s.\mathtt{stdin}$ and $s'.\overline{\mathtt{stdin}} = s.\overline{\mathtt{stdin}}$. Furthermore, $s'.\mu\mathtt{PC}_{sched} = s.\mu\mathtt{PC}_{sched}$. Since π does not reference `stdin`, it also does not reference $\mathtt{lock}_{stdin}$ and $s'.\mathtt{lock}_{stdin} = s.\mathtt{lock}_{stdin}$. Therefore, π_x does not alter the value of any term appearing in implications 7.5.1, 7.5.4 and 7.5.5 and the three implications are maintained.

  Implications 7.5.2 and 7.5.3 are trivially true in s. Furthermore , $s'.\mu\mathtt{PC}_{sched} = s.\mu\mathtt{PC}_{sched}$. Let $s'.\mu\mathtt{PC}_{stdin} = [\pi', x']$. If $\pi = \pi'$ then implications 7.5.2 and 7.5.3 remain trivially true in s'. If $\pi \neq \pi'$ then $x' = 1$ and implication 7.5.3 remains trivially true. Since $x' = 1$, $s'.\mathtt{lock}_{stdin} \neq \mathtt{stdin}$, either $s'.\mathtt{lock}_{stdin} = \mathtt{schedule}$ or $s'.\mathtt{lock}_{stdin} = \mathtt{idle}$. If the latter, implication 7.5.1 implies that $s'.\mathtt{stdin} = s'.\overline{\mathtt{stdin}}$. Either way, implication 7.5.2 is true in s'.

- If π_x is the first micro-action in a locally-controlled micro-action sequence (*i.e.*, $x = 1$) in the stdin thread that references `stdin` then we consider two cases depending on the value of $\mathtt{lock}_{stdin}$. If $s.\mathtt{lock}_{stdin} = \mathtt{schedule}$, then by construction, π_x is a no-op and $s' = s$. So the invariant is maintained. If $s.\mathtt{lock}_{stdin} = \mathtt{idle}$ then, by implication 7.5.1 $s.\mathtt{stdin} = s.\overline{\mathtt{stdin}}$. The effect of the action increments $\mu\mathtt{PC}_{stdin}$ and grabs the lock so $s'.\mu\mathtt{PC}_{stdin} = [\pi, 2]$ and $s'.\mathtt{lock}_{stdin} = \mathtt{stdin}$. Therefore, implications 7.5.1, 7.5.2, and 7.5.4 are trivially true in s'. Let $s'.\mu\mathtt{PC}_{sched} = [\pi', x']$. By invariant 7.1 either π' does not reference `stdin` or $x = 1$. Either way, implications 7.5.5 is true in s'. The effect of the action applies $f_{\pi,1}$ to $s.\mathtt{stdin}$ so implication 7.5.3 remains true in s' because $f_{\pi,x}$ has been applied to both sides of the equality.

- If π is a locally-controlled micro-action sequence in the stdin thread that references `stdin` and π_x is neither the first nor last action in the sequence (*i.e.*, $1 < x < z$), then the implications 7.5.1 and 7.5.2 are trivial while implication 7.5.3 is not. The effect of the action

applies $f_{\pi,x}$ to s. Therefore, $s'.\texttt{stdin} = f_{\pi,x}(s).\texttt{stdin}$ and $s'.\mu\texttt{PC}_{stdin} = [\pi, x+1]$. By construction, $s'.\overline{\texttt{stdin}} = s.\overline{\texttt{stdin}}$, implication 7.5.2 remains trivially true in s' because $x + 1 > 1$. Implication 7.5.3 remains true in s' because $f_{\pi,x}$ has been applied to both sides of the equality.

By Locking Invariant 7.1, since $x > 1$, $s.\texttt{lock}_{stdin} = \texttt{stdin}$ and $s.\mu\texttt{PC}_{sched}$ either equals $[\phi, 1]$ or $[\theta, y]$ where ϕ is an action in the schedule thread that references `stdin` and θ is action that does not reference `stdin`.

If the former case, implication 7.5.4 is trivial and implication 7.5.5 holds because $s.\texttt{lock}_{stdin} = \texttt{stdin}$. Since π is neither the first nor last micro-action in the sequence, π_x changes neither the value of $\texttt{lock}_{stdin}$ nor the value of $\mu\texttt{PC}_{sched}$. So both invariants remain true in s'. In the latter case, implications 7.5.4 and 7.5.5 are trivially true in s and remain so in s' because π_x does not change $\mu\texttt{PC}_{sched}$. Implication 7.5.1 remains trivially true in s'.

- If π_x is the last micro-action in a locally-controlled micro-action sequence in the stdin thread (*i.e.*, $x = z$) that references `stdin` then, by construction, $s'.\overline{\texttt{stdin}} = s'.\texttt{stdin}$ and $s'.\mu\texttt{PC}_{stdin} = [\phi, 1]$ for some micro-action sequence ϕ in the stdin thread. Implications 7.5.1, 7.5.2, and 7.5.4 are true in s' because $s'.\overline{\texttt{stdin}} = s'.\texttt{stdin}$. Implications 7.5.3 and 7.5.5 are trivially true in s' because the index of $s'.\mu\texttt{PC}_{stdin} = 1$.
- The cases where π is an input micro-action sequence in the schedule thread that references `stdin` are symmetric to the three previous cases.

■

Invariant 7.6 (`stdout` Progress) In all reachable states of $\texttt{N}_i$

1. $\texttt{lock}_{stdin} = \texttt{idle} \Rightarrow \texttt{stdout} = \overline{\texttt{stdout}}$,
2. For all locally-controlled action sequences π in the stdout thread of $\mu\hat{\texttt{N}}_i$ that reference `stdout`,

$$\mu\texttt{PC}_{stdin} = [\pi, 1] \Rightarrow \texttt{stdout} = \overline{\texttt{stdout}} \ \vee \ \texttt{lock}_{stdin} = \texttt{schedule},$$

3. For all locally-controlled action sequences π in the stdout thread of $\mu\hat{\texttt{N}}_i$ that reference `stdout` and $1 < x \leq z$,

$$\mu\texttt{PC}_{stdin} = [\pi, x] \Rightarrow f_{\pi,x}(f_{\pi,x-1}(\ldots(f_{\pi,1}(\overline{\texttt{stdout}})))) = \texttt{stdout},$$

4. For all locally-controlled action sequences ϕ in the schedule thread of $\mu\hat{\texttt{N}}_i$ that reference `stdout`,

$$\mu\texttt{PC}_{sched} = [\phi, 1] \Rightarrow \texttt{stdout} = \overline{\texttt{stdout}} \ \vee \ \texttt{lock}_{stdin} = \texttt{stdout}, \text{ and}$$

5. For all locally-controlled action sequences ϕ in the schedule thread of $\mu\hat{\mathtt{N}}_i$ that reference `stdout` and $1 < y \leq z$,

$$\mu\mathtt{PC}_{sched} = [\phi, y] \Rightarrow f_{\phi,y}(f_{\phi,y-1}(\ldots(f_{\phi,1}(\overline{\mathtt{stdout}})))) = \mathtt{stdout}.$$

Proof: The proof of Invariant 7.6 is analogous to that of Invariant 7.5. ■

The Precondition Stability Invariant says that the precondition of the macro-action corresponding to each micro-action sequence (one in each thread) currently scheduled is always true when evaluated on the history variables of $\mu\hat{\mathtt{N}}_i$.

The proof of the following invariant depends explicitly on our assumptions about the behavior of the schedule action and of the actions that enqueue and dequeue elements from the shared buffers `stdin` and `stdout`. In the former case, we assume that the schedule only schedules enabled actions. In the latter case, we use the fact that enqueue action cannot disable the dequeue action. We need this second assumption because the precondition check (presumably) performed by the schedule action before scheduling the dequeue action is not atomic with the actual execution of the action. That is, the enqueue action in one thread can happen between the schedule action and dequeue action in another thread.

Invariant 7.7 (Precondition Stability) In all reachable states of $\mu\hat{\mathtt{N}}_i$,

1. For all locally-controlled action sequences π in the stdin thread of $\mu\hat{\mathtt{N}}_i$,

$$\mu\mathtt{PC}_{stdin} = [\pi, x] \Rightarrow \mathbf{pre}_{\pi}(\overline{\mathtt{PC}}_{stdin}, \overline{\mu\mathtt{ST}}_{stdin}, \overline{\mathtt{stdin}}, \overline{\mathtt{stdout}}),$$

2. For all locally-controlled action sequences ϕ in the stdout thread of $\mu\hat{\mathtt{N}}_i$,

$$\mu\mathtt{PC}_{stdout} = [\phi, y] \Rightarrow \mathbf{pre}_{\phi}(\overline{\mathtt{PC}}_{stdout}, \overline{\mu\mathtt{ST}}_{stdout}, \overline{\mathtt{stdin}}, \overline{\mathtt{stdout}}), \text{ and}$$

3. For all locally-controlled action sequences θ in the schedule thread of $\mu\hat{\mathtt{N}}_i$,

$$\mu\mathtt{PC}_{sched} = [\theta, w] \Rightarrow \mathbf{pre}_{\theta}(\overline{\mathtt{PC}}_{sched}, \overline{\mu\mathtt{ST}}_{sched}, \overline{\mathtt{stdin}}, \overline{\mathtt{stdout}}).$$

Proof: We show Invariant 7.7 by induction on the length of an execution of $\mu\hat{\mathtt{N}}_i$. In the initial state s_0 of $\mu\hat{\mathtt{N}}_i$, $\overline{\mathtt{PC}}_{sched} = \overline{\mathtt{PC}}_{stdin} = \overline{\mathtt{PC}}_{stdout} = \mathtt{schedule}$. In all three threads, the precondition of the schedule action is that the controlling $\mathtt{PC} = \mathtt{schedule}$. So Invariant 7.7 holds.

For the inductive case, we assume the invariant holds in some reachable state s and we show that every micro-action π_x resulting in post-state s' when executed in s maintains the invariant. We proceed by case analysis on the micro-action π_x and the sequence π of which it is a part.

- If π_x is a micro-action in an input sequence in the stdin thread then π is not locally-controlled. Thus, each micro-action in the sequence other than the last trivially maintains implication 7.7.1. The last micro-action π_z in the sequence sets $\mu\texttt{PC}_{stdin}$ to $[\texttt{schedule}, 1]$ and $\overline{\texttt{PC}}_{stdin}$ to `schedule`. Since $\textbf{pre}_{\texttt{schedule}}$ only requires that $\texttt{PC}_{stdin} = \texttt{schedule}$, π_z also maintains implication 7.7.1. By construction, no micro-action in the sequence references $\mu\texttt{PC}_{stdout}$, $\mu\texttt{PC}_{sched}$, $\overline{\texttt{PC}}_{stdout}$, $\overline{\texttt{PC}}_{sched}$, $\overline{\mu\texttt{ST}}_{stdout}$, $\overline{\mu\texttt{ST}}_{sched}$, $\overline{\texttt{stdin}}$, or $\overline{\texttt{stdout}}$ and, therefore, each action in the sequence maintains implications 7.7.2 and 7.7.3.

  Analogous arguments show that the invariant is maintained if π_x is a micro-action in an input sequence in either the stdout or schedule threads.

- If π is a locally-controlled micro-action sequence in the stdin thread and π_x is not the last action in the sequence (*i.e.*, $x < z$), then π_x does not affect history variables, $\mu\texttt{PC}_{stdout}$, or $\mu\texttt{PC}_{sched}$ and, therefore, implications 7.7.2 and 7.7.3 remain true in s'. If $x = 1$ and $\texttt{lock}_{stdin} = \texttt{schedule}$ and sequence π references `stdin` then π_x is a no-op and the invariant is maintained. Otherwise, $s'.\mu\texttt{PC}_{stdin} = [\pi, x + 1]$ and the invariant is maintained.

  Analogous arguments show that the invariant is maintained if π is a locally-controlled sequence in either the stdout or schedule threads and π_x is not the last action in the sequence.

- If π_x is the last micro-action in a locally-controlled micro-action sequence in the stdin thread and π is not the schedule action (*i.e.*, $\pi \neq \texttt{schedule}$ and $x = z$) and π does not reference `stdin` then by construction $s'.\mu\texttt{PC}_{stdin} = [\texttt{schedule}, 1]$ and $s'.\overline{\mu\texttt{PC}} = [\texttt{schedule}]$. Since $\textbf{pre}_{\texttt{schedule}}$ only requires that $\texttt{PC}_{stdin} = \texttt{schedule}$, implication 7.7.1 is maintained. Since π_x does not reference $\mu\texttt{PC}_{sched}$, $\mu\texttt{PC}_{stdout}$, $\overline{\texttt{PC}}_{sched}$, $\overline{\texttt{PC}}_{stdout}$, $\overline{\mu\texttt{ST}}_{sched}$, $\overline{\mu\texttt{ST}}_{stdout}$, $\overline{\texttt{stdin}}$, or $\overline{\texttt{stdout}}$, implication 7.7.2 is maintained.

  If π_x is the last micro-action in a locally-controlled sequence π that does not reference `stdout` in the stdout thread or π_x is the last micro-action in a locally-controlled sequence π that references neither `stdin` nor `stdout` in the schedule thread, analogous arguments show the invariant is maintained.

- If π_x is the last micro-action in a locally-controlled micro-action sequence in the stdin thread and π is not the schedule action (*i.e.*, $\pi \neq \texttt{schedule}$ and $x = z$) but π does reference `stdin` then by construction $s'.\mu\texttt{PC}_{stdin} = [\texttt{schedule}, 1]$ and $s'.\overline{\mu\texttt{PC}} = [\texttt{schedule}]$. Since $\textbf{pre}_{\texttt{schedule}}$ only requires that $\texttt{PC}_{stdin} = \texttt{schedule}$, implication 7.7.1 is maintained. Since π_x does not reference $\mu\texttt{PC}_{stdout}$, $\overline{\texttt{PC}}_{stdout}$, $\overline{\mu\texttt{ST}}_{stdout}$, or $\overline{\texttt{stdout}}$, implication 7.7.2 is maintained.

  Let $s.\mu\texttt{PC}_{sched} = [\theta, w]$. If θ does not reference `stdin` then since π_x does not reference $\mu\texttt{PC}_{sched}$, $\overline{\texttt{PC}}_{sched}$, or $\overline{\mu\texttt{ST}}_{sched}$, implication 7.7.3 is maintained.

  If both π and θ reference `stdin` then π must be the action that enqueues items on `stdin`

(*i.e.*, `appendInvocation`) and θ must be an action that dequeues them. Since π does not reference $\mu\texttt{PC}_{sched}$, $\overline{\texttt{PC}}_{sched}$, $\mathbf{pre}_\theta(s'.\overline{\texttt{PC}}_{sched}, s.'\overline{\mu\texttt{ST}}_{sched}, s'.\overline{\texttt{stdin}}, s.'\overline{\texttt{stdout}})$ can only be false if $\mathbf{pre}_\theta(s.\overline{\texttt{PC}}_{sched}, s.\overline{\mu\texttt{ST}}_{sched}, s'.\overline{\texttt{stdin}}, s.\overline{\texttt{stdout}})$ is also false — that is, if the change to $\overline{\texttt{stdin}}$ falsified the precondition of the dequeuing action. However, by the $\mu\texttt{ST}_{stdin}$ Progress Invariant $s'.\overline{\texttt{stdin}} = s'.\texttt{stdin} = f_{\pi,z}(s).\texttt{stdin} = f^*_\pi(s.)\overline{\texttt{stdin}}$. We assume that the difference between $s.\overline{\texttt{stdin}}$ and $f^*_\pi(s).\overline{\texttt{stdin}}$ is that an element has been added to the tail of $\overline{\texttt{stdin}}$. (Note this is an assumption on f^*_π rather than on s.) The precondition of θ depends only on the head of $\overline{\texttt{stdin}}$. Therefore, π_x cannot falsify $\mathbf{pre}_\theta$ and the invariant is maintained.

Analogous arguments show that the invariant is maintained if π_x is the last micro-action in a locally-controlled sequence in either the stdout or schedule threads.

- Suppose π_x is the last micro-action in the schedule micro-action sequence in the stdin thread (*i.e.*, $\pi =$ `schedule` and $x = z$). Since π_x does not reference $\mu\texttt{PC}_{stdout}$, $\overline{\texttt{PC}}_{stdout}$, $\overline{\mu\texttt{ST}}_{stdout}$, or $\overline{\texttt{stdout}}$, implication 7.7.2 is maintained.

  By construction, $s'.\overline{\texttt{stdin}} = s'.\texttt{stdin} = f_{\pi,z}(s).\texttt{stdin}$. Thus, by the `stdin` Progress Invariant, $s'.\overline{\texttt{stdin}} = f^*_\pi(s).\overline{\texttt{stdin}}$. Since π is the schedule micro-action sequence, we assume that f^*_π is the identity on `stdin`. (NDR programs are not permitted to alter state variables.) Therefore $s'.\overline{\texttt{stdin}} = s.\texttt{stdin}$. Furthermore, since π_x does not reference $\mu\texttt{PC}_{sched}$, $\overline{\texttt{PC}}_{sched}$, $\overline{\mu\texttt{ST}}_{sched}$, or $\overline{\texttt{stdout}}$, implication 7.7.2 is maintained.

  By construction $s'.\mu\texttt{PC}_{stdin} = [\phi, 1]$ where ϕ is some micro-action sequence in the stdin thread. Furthermore, by the definition of the schedule action, the precondition of the selected action, $\mathbf{pre}_{\phi,1}(s'.\mu\texttt{PC}_{stdin}, s'.\mu\texttt{ST}_{stdin}, s'.\texttt{stdin}, s'.\texttt{stdout})$ is true in s'. By the $\texttt{ST}_{stdin}$ Progress Invariant $s'.\mu\texttt{ST}_{stdin} = s'.\overline{\mu\texttt{ST}}_{stdin}$ and $s'.\overline{\texttt{PC}}_{stdin} = \phi$. Since, the only difference between $\mathbf{pre}_{\phi,1}$ and $\mathbf{pre}_\phi$ is that the former requires $\mu\texttt{PC}_{stdin} = [\phi, 1]$ while the latter requires $\texttt{PC} = \phi$, the invariant is maintained.

  Analogous arguments show that the invariant is maintained if π_x is the last micro-action in the schedule sequence in either the stdout or schedule threads.

■

7.3.6 Refinement Mapping

We prove Theorem 7.1 by showing a refinement mapping $\mathcal{M}$ from $\mu\hat{\texttt{N}}_i$ to $\texttt{N}_i$. If s is a state of $\mu\hat{\texttt{N}}_i$ then we define $S = \mathcal{M}(s)$ to be a state of $\texttt{N}_i$ where

1. $S.\texttt{ST}_{sched} = s.\overline{\mu\texttt{ST}}_{sched}$,
2. $S.\texttt{ST}_{stdin} = s.\overline{\mu\texttt{ST}}_{stdout}$,

3. $S.\mathtt{ST}_{stdout} = s.\overline{\mu\mathtt{ST}}_{stdout}$,

4. $S.\mathtt{stdin} = s.\overline{\mathtt{stdin}}$,

5. $S.\mathtt{stdout} = s.\overline{\mathtt{stdout}}$,

6. $S.\mathtt{PC}_{sched} = s.\overline{\mathtt{PC}}_{sched}$,

7. $S.\mathtt{PC}_{stdin} = s.\overline{\mathtt{PC}}_{stdin}$, and

8. $S.\mathtt{PC}_{stdout} = s.\overline{\mathtt{PC}}_{stdout}$.

In this refinement mapping, the history variables are used to delay or accelerate the effects of an entire micro-action sequences to the point in the execution where the externally visible action occurs. Thus, each externally visible micro-action appears to have the same effect as the corresponding macro-action happening at the same point in the execution. The effects of a sequence for a locally-controlled action are delayed until the end of the sequence. The effects of a sequence for an input action are accelerated to the beginning of the sequence. In other words, it appears to an external observer as if the execution of $\mu\hat{\mathtt{N}}_i$ had been reordered so that every micro-action sequence happens all in one step, uninterrupted by any interleaving.

Applying the definition of a refinement mapping from Section 2.1.2, $\mathcal{M}$ is a refinement mapping if: (1) for any initial state s_0 of $\mu\hat{\mathtt{N}}_i$, $S_0 = \mathcal{M}(s_0)$ is an initial state of $\mathtt{N}_i$, and (2) for any reachable states s and $S = \mathcal{M}(s)$ of $\mu\hat{\mathtt{N}}_i$ and $\mathtt{N}_i$, respectively, and for any transition π of $\mu\hat{\mathtt{N}}_i$ enabled in state s and resulting in state s', there is a (possibly empty) sequence of transitions α of $\mathtt{N}_i$ that results in state $S' = \mathcal{M}(s')$ where α has the same trace as π. Figure 7.9 depicts the requirements schematically.

The proof that $\mathcal{M}$ is a refinement mapping from $\mu\hat{\mathtt{N}}_i$ to $\mathtt{N}_i$ proceeds by proving two lemmas. Lemma 7.3 asserts the initial state correspondence holds. Lemma 7.4 asserts a step correspondence maintains the state correspondence.

Lemma 7.3 If s_0 is an initial state of $\mu\hat{\mathtt{N}}_i$ then $\mathcal{M}(s_0)$ is an initial state of $\mathtt{N}_i$.

Proof: Let s_0 be the unique initial state of $\mu\hat{\mathtt{N}}_i$ and S_0 be the unique initial state of $\mathtt{N}_i$. By definition, in S_0 all variables of $\mathtt{N}_i$ have the same values as the corresponding history variables of $\mu\hat{\mathtt{N}}_i$ in s_0. In the unique initial states of $\mu\hat{\mathtt{N}}_i$ and $\mathtt{N}_i$, $\mathtt{ST}_{sched} = \mu\mathtt{ST}_{sched}$, $\mathtt{ST}_{stdin} = \mu\mathtt{ST}_{stdin}$, $\mathtt{ST}_{stdout} = \mu\mathtt{ST}_{stdout}$, and $\overline{\mathtt{PC}}_{sched} = \overline{\mathtt{PC}}_{stdin} = \overline{\mathtt{PC}}_{stdout} = \mathtt{PC}_{sched} = \mathtt{PC}_{stdin} = \mathtt{PC}_{stdout} = \mathtt{schedule}$. Since three history variables, $\overline{\mu\mathtt{ST}}_{sched}$, $\overline{\mu\mathtt{ST}}_{stdin}$, and $\overline{\mu\mathtt{ST}}_{stdout}$ are initialized to the initial values of $\mu\mathtt{ST}_{sched}$, $\mu\mathtt{ST}_{stdin}$, and $\mu\mathtt{ST}_{stdout}$ the unique initial state of $\mu\hat{\mathtt{N}}_i$ is $\mathtt{S}_0$, as needed. ■

Lemma 7.4 Let s be a reachable state of $\mu\hat{\mathtt{N}}_i$, $S = \mathcal{M}(s)$ be a reachable state of $\mathtt{N}_i$, and π_x be a transition of $\mu\hat{\mathtt{N}}_i$ resulting in state s' when executed in s. There is a sequence of transitions α of $\mathtt{N}_i$, enabled in S, that results in state $S' = \mathcal{M}(s')$, such that $trace(\alpha) = trace(\pi)$.

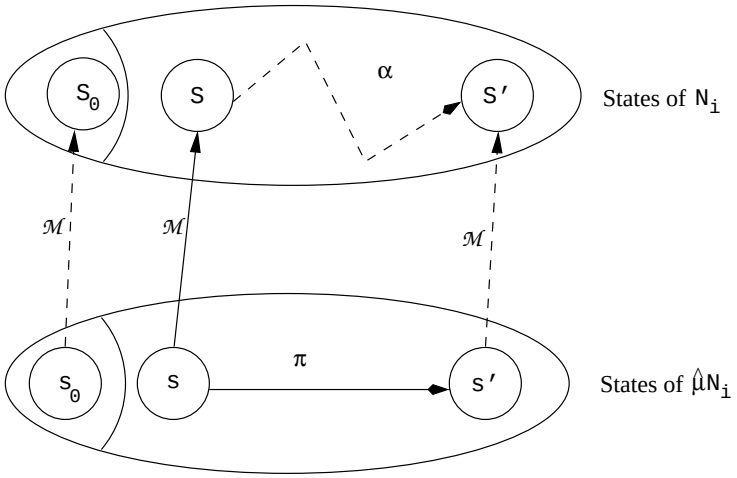

Figure 7.9: Schematic of system refinement mapping $\mathcal{M}$. $\mathcal{M}$ is a refinement mapping from $\mathtt{N}_i$ to $\mu\hat{\mathtt{N}}_i$ if (1) for any initial state s_0 of $\mu\hat{\mathtt{N}}_i$, $S_0 = \mathcal{M}(s_0)$ is an initial state of $\mathtt{N}_i$, and (2) for any reachable states s and $S = \mathcal{M}(s)$ of $\mu\hat{\mathtt{N}}_i$ and $\mathtt{N}_i$, respectively, and for any transition π of $\mu\hat{\mathtt{N}}_i$ enabled in state s and resulting in state s', there is a sequence of transitions α of $\mathtt{N}_i$ that results in state $S' = \mathcal{M}(s')$ where α has the same trace as π.

Proof: We prove Lemma 7.4 by case analysis on the micro-action π_x of $\mu\hat{\mathtt{N}}_i$ and the corresponding macro-action sequences α of $\mathtt{N}_i$. Let $S = \mathcal{M}(s)$ and $S' = \mathcal{M}(s')$.

If π_x is neither the first nor last step in a micro-action sequence, α is empty. Similarly, if π_x the last step of an input sequence or the first step on an output sequence, α is empty. In none of these cases do either the history variables of $\mu\hat{\mathtt{N}}_i$ or the state of $\mathtt{N}_i$ change. Therefore, α results in state $S = \mathcal{M}(s) = \mathcal{M}(s') = S'$, as needed. Since π must be an internal action in these cases, both $trace(\pi_x)$ and $trace(\alpha)$ are empty, as needed.

The interesting cases are when π_x begins an input micro-action sequence or ends a locally-controlled action sequence. That is, π_x is either an input micro-action π_1 derived from an input macro-action ϕ, an internal or output micro-action π_z derived from an internal or output macro-action π. In those cases, α is the macro-action π of $\mathtt{N}_i$ from which π_x is derived.

- Suppose π_x is the first micro-action in an input sequence in the stdin thread (*i.e.*, $x = 1$). Since micro-action π_1 is derived from input macro-action π, $trace(\pi_x) = trace(\alpha) = \pi$, as needed. Since $S = \mathcal{M}(s)$, all variables in S are equal to the corresponding history micro-variables in s. Since π_x does not reference $\overline{\mathtt{PC}}_{sched}$, $\overline{\mathtt{PC}}_{stdout}$, $\overline{\mu\mathtt{ST}}_{sched}$, $\overline{\mu\mathtt{ST}}_{stdout}$, $\overline{\mathtt{stdin}}$, $\overline{\mathtt{stdout}}$, those variables are unchanged from s to s'. Since the macro-action π does not reference $\mathtt{PC}_{sched}$, $\mathtt{PC}_{stdout}$, $\mathtt{ST}_{sched}$, $\mathtt{ST}_{stdout}$, `stdin`, or `stdout`, those variables are unchanged from S to S'. Thus, the correspondence of these variables is maintained. By $\mu\mathtt{ST}_{stdin}$ Progress Invariant 7.2.1, $s.\mu\mathtt{ST}_{stdin} = s.\overline{\mu\mathtt{ST}}_{stdin}$. The micro-action assigns to $\overline{\mu\mathtt{ST}}_{stdin}$ the result of applying f^*_π to $\mu\mathtt{ST}_{stdin}$ and assigns `schedule` to $\overline{\mathtt{PC}}_{stdin}$. The macro-action updates $\mathtt{ST}_{stdin}$ by applying f_π to

it and sets $\texttt{PC}_{stdin}$ to `schedule`. Since $s.\mu\texttt{ST}_{stdin} = S.\texttt{ST}_{stdin}$, $f^*_\pi(s).\mu\texttt{ST}_{stdin} = s'.\overline{\mu\texttt{ST}}_{stdin} = f_\pi(S).\texttt{ST}_{stdin} = S'.\texttt{ST}_{stdin}$. Finally, $s'.\overline{\texttt{PC}}_{stdin} = S'.\texttt{PC}_{stdin}$. Therefore $S' = \mathcal{M}(s')$, as needed.

- Suppose π_x is the last micro-action in an internal sequence in the stdin thread (*i.e.*, $x = z$). Since micro-action π_x is derived from an internal action both $trace(\pi_x)$ and $trace(\alpha)$ are empty, as needed. Since $S = \mathcal{M}(s)$, all variables in S are equal to the corresponding history micro-variables in s. By the Precondition Stability Invariant 7.7, π is enabled in S.

  Since ϕ_x does not reference $\overline{\texttt{PC}}_{sched}$, $\overline{\texttt{PC}}_{stdout}$, $\overline{\mu\texttt{ST}}_{sched}$, or $\overline{\mu\texttt{ST}}_{stdout}$ those variables are unchanged from s to s'. Since the macro-action ϕ does not reference $\texttt{PC}_{sched}$, $\texttt{PC}_{stdout}$, $\texttt{ST}_{sched}$, or $\texttt{ST}_{stdout}$ those variables are unchanged from S to S'. Thus, the correspondence of these variables is maintained. The micro-action assigns to $\overline{\mu\texttt{ST}}_{stdin}$ the result of applying $f_{\phi,z}$ to $\mu\texttt{ST}_{stdin}$ and to $\overline{\texttt{stdin}}$ the result of applying $f_{\phi,z}$ to `stdin`. The macro-action updates $\texttt{ST}_{stdin}$ and `stdin` by applying f_ϕ to them. By $\mu\texttt{ST}_{stdin}$ Progress Invariant 7.2.3 and `stdin` Progress Invariant 7.5.1, $s'.\mu\texttt{ST}_{stdin} = s.\overline{\mu\texttt{ST}}_{stdin}$ and $s'.\texttt{stdin} = s'.\overline{\texttt{stdin}}$. Since $s.\mu\texttt{ST}_{stdin} = S.\texttt{ST}_{stdin}$, $f_{\phi,z}(s).\mu\texttt{ST}_{stdin} = s'.\overline{\mu\texttt{ST}}_{stdin} = f_\phi(S).\texttt{ST}_{stdin} = S'.\texttt{ST}_{stdin}$. Similarly, $s'.\overline{\texttt{stdin}} = S.\texttt{stdin}$. Finally, by construction, the first element of $s'.\mu\texttt{PC}_{stdin} = s'.\overline{\texttt{PC}}_{stdin}$ equals $S'.\texttt{PC}_{stdin}$. Therefore $S' = \mathcal{M}(s')$, as needed.

- If π is the last micro-action ϕ_z in an output sequence, the argument that α is enabled in S and that $S' = \mathcal{M}(s')$ are analogous to the previous case. Since micro-actions ϕ_z is derived from an output macro-action ϕ, $trace(\pi) = trace(\alpha) = \phi$, as needed.

Analogous arguments show that $S' = \mathcal{M}(s')$ and $trace(\pi) = trace(\alpha)$ if π is a micro-action ϕ_x in the schedule or stdout threads. ■

Lemma 7.5 The set of traces of $\mu\hat{\texttt{N}}_i$ is a subset of the set of trace of $\texttt{N}_i$.

Proof: It follows immediately from Lemmas 7.3 and 7.4 and the definition of a refinement mapping that $\mathcal{M}$ is a refinement mapping from $\mu\hat{\texttt{N}}_i$ to $\texttt{N}_i$. Lemma 7.5, therefore, follows from Theorem 3.4 of [83] which implies that it is sufficient to show a refinement mapping to demonstrate trace inclusion. ■

Lemma 7.6 The set of traces of $\mu\texttt{N}_i$ is a subset of the set of trace of $\mu\hat{\texttt{N}}_i$.

Proof: Lemma 7.6 follows immediately from Theorem 5.5 of [83] which states that set of traces of an automaton augmented by history is a subset of the traces of the unaugmented automaton. ■

Finally, we are ready to prove Theorem 7.1 that we claimed in Section 7.3.1.

Proof of Theorem 7.1 (Node Correctness): Theorem 7.1 follows from Lemmas 7.5 and 7.6 by transitivity. ■

7.4 Handshake Theorem

As we describe in Section 3.4, we allow the programmer to specify an arbitrary interface to the console while also requiring IOA systems submitted for compilation to be correct even when each node automaton is composed with a buffer automaton. The buffer automata we describe in Section 3.4 implement a handshake protocol similar to the one described in Section 3.3. Unfortunately, this protocol changes the interface between the IOA program and its environment. Since we want to allow the programmer to specify the interface, we would like to relax this obligation.

In this section, we show that augmenting the interface between two automata to follow such a handshake protocol does not substantially change their interaction. More importantly, we show that omitting such a protocol from the interface also does not substantially change the interaction. Formally, we show that, ignoring the added `ready` actions, the set of traces of the system does not change when the handshake augmentations are added or removed. The key insight is that the system performing the handshake can always mimic the behavior of the original system by immediately preceding each input action with the output action to signal that it is ready and following it with whatever local work needs to be done. On the other hand, the original system can always mimic the handshake system because it does not use the ready output signal.

Let `Buf` be a buffer automaton as described in Section 3.4.1 unaugmented by the handshake protocol. That is, omitting the actions `inReady` and `outReady` and the state variables `enabled` and `signalled`. Similarly, let `Env` be the unaugmented environment automaton. Let $\phi_1, \phi_2, \ldots$ be the console input actions of the algorithm automaton from which `Buf` is derived and, consequently, also input actions `Buf` receives from the environment. Similarly, let $\theta_1, \theta_2, \ldots$ be the console output actions of the algorithm automaton from which `Buf` is derived and consequently also output actions `Buf` emits to the environment. Let $\widehat{\texttt{Buf}}$ and $\widehat{\texttt{Env}}$ be the automata with the handshake protocols. For example, while Figure 3.4 show the augmented buffer automaton for the LCR example, Figure 7.10 shows the somewhat simpler automaton the LCR programmer may use to prove correctness.

Let $\texttt{Env} \circ \texttt{Buf}$ be the composition of automata `Env` and `Buf`. Similarly, let $\widehat{\texttt{Buf}}$ be the augmented version of the Let $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ be the composition of augmented automata where the `inReady` and `outReady` actions are hidden. Theorem 7.7 says the externally visible behavior of the system (other than the appearances of `inReady` and `outReady` actions) does not change by adding or removing the handshake protocol between the two components.

Theorem 7.7 (Handshake) The set of traces of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ is equal to the set of traces of $\texttt{Env} \circ \texttt{Buf}$.

We prove Theorem 7.7 by showing two refinement mappings, each showing trace containment in one direction. We define $\mathcal{R}$ which maps the states of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ to the states of $\texttt{Env} \circ \texttt{Buf}$ as follows. If s is a state of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ then $S = \mathcal{R}(s)$ is the projection of s on the state space of $\texttt{Env} \circ \texttt{Buf}$. In other words, $\mathcal{R}$ is the identity, ignoring the `enabled` and `signalled` variables in both $\widehat{\texttt{Buf}}$ and $\widehat{\texttt{Env}}$.

```
type IOA_Invocation = tuple of action: IOA_Action, params: Seq[IOA_Parameter]
type IOA_Parameter = union of Int: Int
type IOA_Action = enumeration of leader, vote
automaton LCRProcessInterface(i: Int, ringSize: Int, name: Int)
  signature
    input
      vote(I0: Int) where I0 = i,
      leader(I5: Int) where I5 = i,
    output
      leader(I5: Int) where I5 = i,
      vote(I0: Int) where I0 = i,
    internal
      appendInvocation(I0: Int) where I0 = i
  states
    valid: Bool := false,
    invocation: IOA_Invocation,
    stdin: LSeqIn[IOA_Invocation]:= {},
    stdout: LSeqOut[IOA_Invocation]:= {}
  transitions
    input vote(I0)
      eff invocation := [vote, {} ⊢ Int(I0)];
          valid := true;
    internal appendInvocation(I0)
      pre valid
      eff stdin := stdin ⊢ invocation;
          valid := false;
    output vote(I0)
      pre stdin ≠ {} ∧
          ((((head(stdin).action) = vote) ∧
          (len(head(stdin).params)) = 1) ∧
          (tag(head(stdin).params[0])) = Int) ∧
          (head(stdin).params[0].Int) = I0
      eff stdin := tail(stdin)
    input leader(I5)
      eff stdout := stdout ⊢ [leader, ({}) ⊢ Int(I5)]
    output leader(I5)
      pre stdout ≠ {} ∧
          (((head(stdout).action) = leader) ∧
          (len(head(stdout).params)) = 1) ∧
          (tag(head(stdout).params[0])) = Int) ∧
          (head(stdout).params[0].Int) = I0
      eff stdout := tail(stdout);
```

Figure 7.10: IOA specification the `LCRProcessInterface` buffer automaton without handshake protocol

We define $\mathcal{M}$ which maps the states of $\texttt{Env} \circ A$ to the states of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ as follows. If s is a state of $\texttt{Env} \circ \texttt{Buf}$ then $S = \mathcal{M}(s)$ is the projection of s on the state space of $\texttt{Env} \circ \texttt{Buf}$ where $\texttt{Buf.enabled} = \texttt{Buf.signalled} = \texttt{Env.enabled} = \texttt{Env.signalled} =$ false. In other words, $\mathcal{M}$ is the identity supplemented by the constant false function of `enabled` and `signalled`.

The proof that each of these functions is a refinement mapping proceeds by proving two lemmas for each. Lemmas 7.8 and 7.9 assert the initial state correspondences hold. Lemmas 7.10 and 7.11 assert that step correspondences maintain the state correspondences.

Lemma 7.8 If s_0 is an initial state of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ then $\mathcal{R}(s_0)$ is an initial state of $\texttt{Env} \circ \texttt{Buf}$.

Proof: Let s_0 be the unique initial state of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ and $S_0 = \mathcal{R}(s_0)$. By definition, in S_0 all variables of $\texttt{Env} \circ \texttt{Buf}$ are equal to the corresponding variables in s_0. In the unique initial state of $\texttt{Env} \circ \texttt{Buf}$, all variables that exist in that automaton are also equal to the corresponding variables of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ in s_0, as needed. ■

Lemma 7.9 If s_0 is an initial state of $\texttt{Env} \circ \texttt{Buf}$ then $\mathcal{M}(s_0)$ is an initial state of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$.

Proof: Let s_0 be the unique initial state of $\texttt{Env} \circ \texttt{Buf}$ and $S_0 = \mathcal{M}(s_0)$. By definition, in S_0 all variables are equal to the corresponding variables in s_0 and `enabled` and `signalled` are false. In the unique initial state of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$, all variables are also equal to the corresponding variables in s_0 and `Buf.enabledf` and `Buf.signalled`, `Env.enabled` and `Env.signalled` are also false, as needed. ■

Lemma 7.10 Let s be a reachable state of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$, $S = \mathcal{R}(s)$ be a reachable state of $\texttt{Env} \circ \texttt{Buf}$, and π be a transition of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$ resulting in state s' when executed in s. There is a sequence of transitions α of $\texttt{Env} \circ \texttt{Buf}$, enabled in S, that results in state $S' = \mathcal{R}(s')$, such that $trace(\alpha) = trace(\pi)$.

Proof: We prove Lemma 7.10 by case analysis of the transitions π of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$. Let $S = \mathcal{R}(s)$.

π **is** `inReady` Let α be $\{\}$. Since `inReady` is hidden, both the trace of π and the trace of α are empty, as needed. Since $S = \mathcal{R}(s)$ and `inReady` affects only `Env.enabled` and $\mathcal{R}$ ignores that variable, $S' = \mathcal{R}(s')$, as needed.

π **is** `outReady` Let α be $\{\}$. Since `outReady` is hidden, both the trace of π and the trace of α are empty, as needed. Since $S = \mathcal{R}(s)$ and `outReady` affects only `Buf.enabled` and `Env.signalled` and $\mathcal{R}$ ignores those variables , $S' = \mathcal{R}(s')$, as needed.

π **is** ϕ_i Let α be $\{\phi_i\}$. The trace of ϕ_i equals the trace of α, as needed. Since $S = \mathcal{R}(s)$, ϕ_i is enabled in s and the precondition of α less strict than that of ϕ_i, α is enabled in S. Since ϕ_i and α have the same effect, as needed.

π **is** `appendInvocation` Let α be $\{$`appendInvocation`$\}$. The trace of θ_i equals the trace of α, as needed. Since $S = \mathcal{R}(s)$, θ_i is enabled in s and the precondition of α identical to that of π, α is enabled in S. Since π and α have the same effect other than assignments to `Buf.signalled` and since $\mathcal{R}$ ignores that variable, $S' = \mathcal{R}(s')$, as needed.

π **is any other action** Let α be $\{\pi\}$. The trace of π equals the trace of α, as needed. Since $S = \mathcal{R}(s)$ and π is enabled in s, α is enabled in S. Since π and α have the same effect, $S' = \mathcal{R}(s')$, as needed.

■

Lemma 7.11 Let s be a reachable state of `Env`$\circ$`Buf`, $S = \mathcal{M}(s)$ be a reachable state of $\widehat{\texttt{Env}}\circ\widehat{\texttt{Buf}}$, and π be a transition of `Env`$\circ$`Buf` resulting in state s' when executed in s. There is a sequence of transitions α of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$, enabled in S, that results in state $S' = \mathcal{M}(s')$, such that $trace(\alpha) = trace(\pi)$.

Proof: We prove Lemma 7.11 by case analysis of the transitions π of `Env`$\circ$`Buf` and the corresponding transition sequences α of $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$. Let $S = \mathcal{M}(s)$.

π **is** ϕ_i Let α be $\{$`inReady`$, \phi_i,$ `appendInvocation`$\}$. The trace of π equals the trace of α, as needed. Since $S = \mathcal{M}(s)$, all four `enabled` and `signalled` variables are false in S and, therefore, `inReady` is enabled in S. After the execution of `inReady`, both $\widehat{\texttt{Env}}$`.enabled` and $\widehat{\texttt{Buf}}$`.signalled` are true, so ϕ_i is enabled. Execution of ϕ_i toggles $\widehat{\texttt{Buf}}$`.valid` to true and $\widehat{Env}$`.enabled` to false, enabling `appendInvocation`. Thus, α is enabled in S. Since ϕ_i and α have the same effect other than assignments to $\widehat{Env}$`.enabled` and $\widehat{Buf}$`.signalled` and since both those are assigned false by `inReady` and `appendInvocation`, respectively, $S' = \mathcal{M}(s')$, as needed.

π **is** θ_i Let α be $\{$`outReady`$, \theta_i\}$. The trace of π equals the trace of α, as needed. Since $S = \mathcal{M}(s)$, all four `enabled` and `signalled` variables are false in S and, therefore, `outReady` is enabled in S. After the execution of `outReady`, both $\widehat{\texttt{Buf}}$`.enabled` and $\widehat{\texttt{Env}}$`.signalled` are true, and, thus, θ_i is enabled. Execution of θ_i toggles both those variables back to false. Since θ_i and α have the same effect other than the assignments to those two variables, $S' = \mathcal{M}(s')$, as needed.

π **is any other action,** $\alpha = \pi$ The trace of π equals the trace of α, as needed. Since $S = \mathcal{R}(s)$, π is enabled in s, α is enabled in S. Since π and α have the same effect, $S' = \mathcal{M}(s')$, as needed.

■

Finally we are ready to prove Theorem 7.7.

Proof of Theorem 7.7 (Handshake): Lemmas 7.8 and 7.10, and the definition of a refinement mapping that $\mathcal{R}$ is a refinement mapping from $\widehat{\texttt{Env}}\circ\widehat{\texttt{Buf}}$ to `Env`$\circ$`Buf`. Similarly, Lemmas 7.9 and 7.11 imply that $\mathcal{M}$ is a refinement mapping from `Env`$\circ A$ to $\widehat{\texttt{Env}} \circ \widehat{\texttt{Buf}}$. Theorem 7.7 then follows from from Theorem 3.4 of [83]. ■

Chapter 8

Experimental Evaluation

We are, I think, in the right Road of Improvement, for we are making Experiments.

— Benjamin Franklin [42]

To evaluate the IOA compiler and the code it produces, we have written IOA automata to implement three distributed algorithms from the literature. We have compiled these node automata and run the resulting systems to demonstrate the functionality of the generated code, measure some its basic properties, and make some observations about the compilation process. Measuring the performance of the running algorithms establishes some quantitative benchmarks for comparing this prototype to any alternative implementations or future optimizations. The three algorithms we have used are LCR leader election, computation of a spanning tree, and repeated broadcast/convergecast over a computed spanning tree [75, 18, 19, 107].

Our selected experiments exercise many features of the compiler. First and foremost, we show that distributed message-passing algorithms run and pass messages as expected. In doing so, we employ most of the catalog of IOA datatypes, datatype constructors, and record types. The basic datatypes are booleans, integers, natural numbers, characters, and strings. The type constructors are arrays, sets, multisets, sequences, and mappings. The record types are enumerations, tuples, and unions. Of these, we use all but naturals, characters, and strings. In addition, we introduce new and enhanced datatypes not in the basic IOA language. For example, we enhance the `Set` and `Mset` datatype constructors with choice operators and introduce a `Null` type constructor. We demonstrate the use of all the supported control structures in the language including loops that iterate over the elements of finite sets.

We show the initialization of automaton formal parameters at run time both from user provided

input and from values extracted from the MPI runtime environment. We use this functionality to specify and to adapt to the topology and size of the network on which the systems run. We run algorithms on a variety of network sizes and topologies.

Below we discuss our experimental testbed, our measurements of each algorithm, and some general observations about the compiler.

8.1 Testbed

Our experimental testbed consists of a collection of networked workstations. Each machine has a direct wire connection to a common Ethernet switch. The macines in our study are built with Pentium III CPUs running at clock speeds of 2–3.2GHz using 128–896 megabytes of RAM. Each operates under Red Hat Linux release 9 (Shrike). We use Sun Microsystems' Java 2 SDK, Standard Edition Version 1.4.2. Our MPI implementation is mpiJava Version 1.2.5 from Indiana University running over mpich 1.2.5.2 from Argonne National Laboratory.

We have enhanced the compiler to output code instrumented to measure the algorithm runtime at each node in the system. We exclude startup and initialization from our timing measurements by having each node perform an MPI **Barrier** call after initialization but before executing any transitions or processing any input. The **Barrier** call does not return until every node in the system has executed the call. The timing period ends after the main thread completes its schedule. Each node records its elapsed run time, the number of transitions it executed and the number of executions of each of the four MPI messaging calls (**Isend**, **test**, **Iprobe**, and **receive**).

MPI requires that no node shuts down its MPI service while other nodes in the system are continuing to use MPI. Therefore, each node performs a second **Barrier** call before exiting. In our experiments, we used only algorithms that terminate at every node.

8.2 LCR Leader Election

In our first experiment, we generated code for the LeLann-Chang-Roberts (LCR) leader election algorithm. We modified the algorithm presented in earlier chapters to achieve termination at all nodes rather than just at the leader. Termination at every node is achieved by adding an extra communication round in which the leader announces its status to the other nodes by sending a message around the ring as well as to the outside world by executing an output action. Each node schedule terminates after the announcement message has been forwarded. The algorithm automaton for the terminating version of LCR is shown in Figure 8.1. The code for the composed, scheduled, terminating version of LCR appears in Appendix A.2.

```
type Status = enumeration of idle, voting, elected, announced, over

automaton TermininatingLCRProcess(i, ringSize, name: Int)
  signature
    input vote(const i)
    input RECEIVE(m: Int, const mod(i-1, ringSize), const i)
    output SEND(m: Int, const i, const mod(i+1, ringSize))
    output leader(const i)

  states
    pending: Mset[Int] := {name},
    status: Status := idle

  transitions
    input vote(i)
      eff status := voting
    input RECEIVE(m, j, i) where m > name
      eff pending := insert(m, pending)
    input RECEIVE(m, j, i) where 0 ≤ m ∧ m < name
    input RECEIVE(m, j, i) where m < 0
      eff if status ≠ announced then
            pending := insert(m, pending)
          fi;
          status := over
    input RECEIVE(name, j, i)
      eff status := elected
    output SEND(m, i, j)
      pre status ≠ idle ∧ m ∈ pending
      eff pending := delete(m, pending)
    output leader(i)
      pre status = elected
      eff status := announced;
          pending := insert(-1, pending)
```

Figure 8.1: Algorithm automaton `TerminatingLCRProcess` specifies an LCR process where every node knows when the leader has been announced.

NODES	SAMPLES	MESSAGES	TRANSITIONS	IPROBES	RUNTIME (σ)
2	962	5.00	19.3	5.30	0.158 (0.030)
3	966	8.80	25.4	8.60	0.227 (0.048)
4	961	12.7	30.7	11.8	0.308 (0.069)
5	966	16.0	32.4	13.2	0.325 (0.066)
6	931	20.2	37.5	17.0	0.418 (0.13)
7	905	24.9	41.5	19.9	0.485 (0.23)
8	923	29.3	48.1	25.2	0.603 (0.26)
9	902	34.9	51.0	26.5	0.633 (0.23)
10	728	37.3	50.0	28.5	0.662 (0.20)

Table 8.1: Measurements of LCR. The columns are the number of the number of nodes in the ring, the number of sample runs measured, the total number of messages sent across all channels in a run, the maximum number of transitions executed at any node in a run, the maximum number of `Isend`s performed at any node in a run, and the maximum runtime at any node on a run. Figures shown in the last four columns are averages across all sample runs. The standard deviation of the maximum runtime is shown in parenthesis. Runtimes times are measured in seconds.

8.2.1 Results

We ran LCR on rings ranging in size from two to ten nodes. For each ring size we executed one thousand runs of LCR in batches of one hundred. We permuted the names of the nodes randomly before each batch. The number of messages sent by the algorithm is wholly determined by the permutation of the names. We spot checked the correctness of the result. As expected, in every case, only the node with the largest name executed the `leader` output action.

Table 8.1 summarizes our measurements. We take the maximum runtime at any node as the runtime of the algorithm. Similarly, we report the maximum number of transitions and **Iprobe** calls executed at any node in a particular node. For messages, we report the total number of messages sent by all nodes in the ring for that run. Each row in the column contains information about the one thousand runs for a particular ring size. The latter four columns show the averages of these measurements over the measured runs. The standard deviation of the runtimes is shown in parenthesis.

In the first eight rows of Table 8.1, we have excluded runs where the runtime exceeded the mean by more than two standard deviations. In the case of the ten-node rings we have excluded any runs where the runtime exceeded the mean by more than three quarters of a standard deviation. The reason for this broad definition of an outlier is that the samples show a distinctly bimodal distribution. (This large outlier set also accounts for the substantially smaller number of samples included.) Approximately ten percent of all runs involving six nodes exhibit excessive runtimes. The distribution in the ten node case was so broad that we had to expand our definition of outlier to

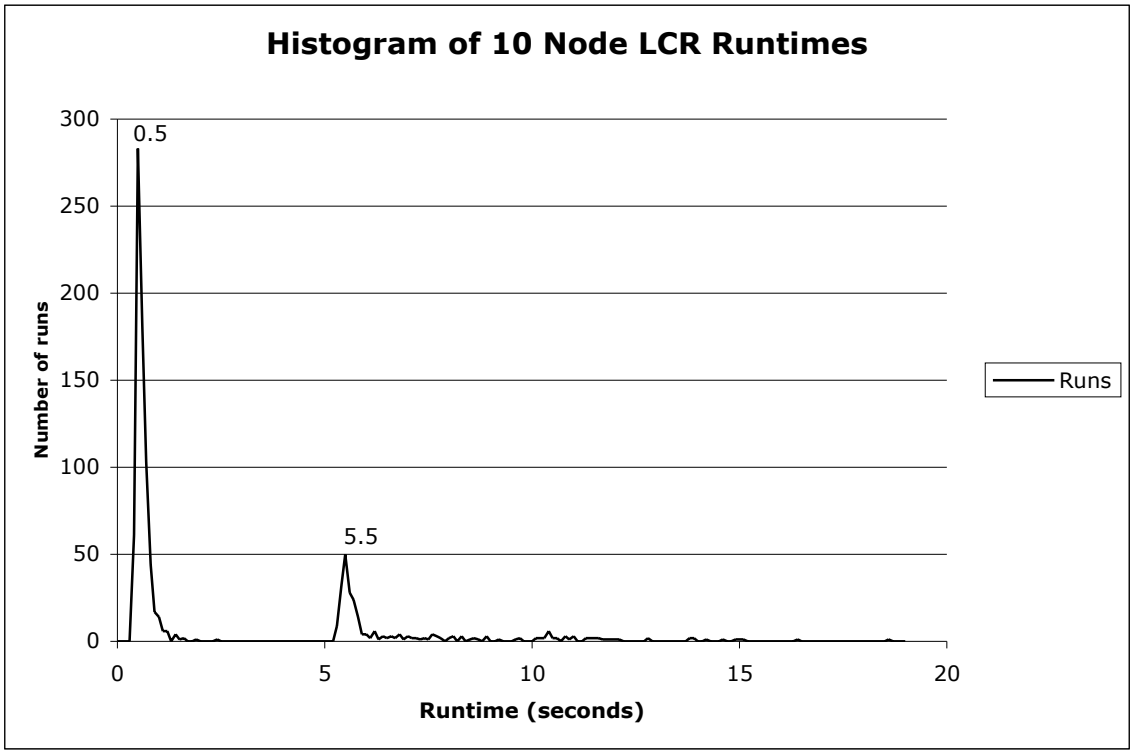

Figure 8.2: 10 Node LCR Histogram

separate the two apparent distributions. We have yet to find the explanation for this phenomenon but we suspect some artifact of our MPI installation. For all five larger node sizes, the larger mode was at a runtime approximately five seconds longer than the reported mean. Figure 8.2 show a histogram of all one thousand measured runtimes for LCR on a ten-node ring.

Figure 8.3 plots the four means against the number of nodes in the ring. The runtimes are plotted against the left hand Y-axis showing the time in seconds. The other events are counted on the right hand Y-axis. We observe that the all four measurements increase linearly with the number of nodes in the ring, as expected. The number of transitions executed is dominated by the number of **Iprobe** calls performed. (Although each MPI call corresponds to two IOA transitions in the model, we count only the output in these measurements.) There is a substantial opportunity for tuning the system performance by adjusting the ratio of **Iprobe** calls to messages actually received. For example, following a suggestion by Mavrommatis and Georgiou based on their preliminary experiments, we improved system performance by more than an order of magnitude by injecting sleep into the thread schedules [87]. The main thread sleeps ten milliseconds after each **Iprobe** and each of the input and output threads sleeps ten milliseconds after each iteration of its schedule loop. Measured run times fell from more than ten seconds of elapsed time on a ten-node ring to less than half a second.

Figure 8.4 shows a sample of an execution of LCR. Command line arguments control whether such transition by transition records are generated at runtime. No such records were produced in

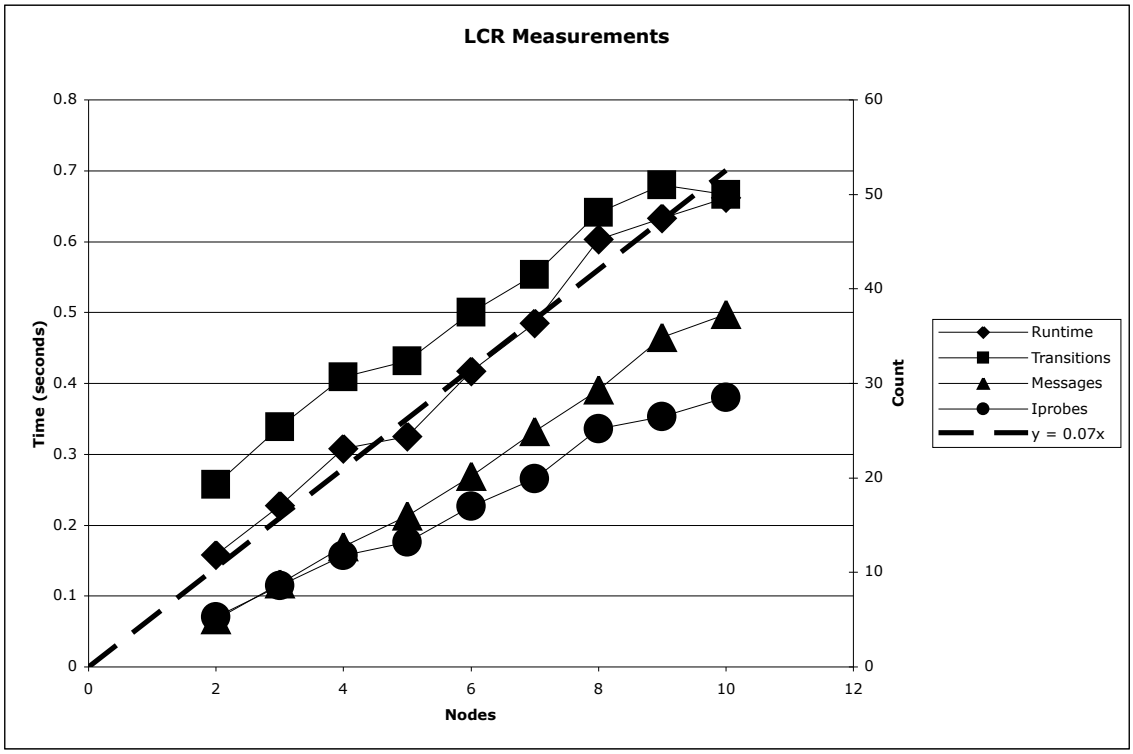

Figure 8.3: LCR Measurements

the measured runs but output actions were generated. The sequence of six transitions in that figure shows node 4 processing a message "9" from node 3. In this run, the name of node 4 is "2", so the node forwards the message to node 5.

8.3 Spanning Tree

In our second experiment, we implemented the $AsynchSpanningTree_i$ algorithm presented in Section 15.3 of [81]. Our automaton is based on an implementation of the algorithm by Mavrommatis and Georgiou [88]. Figure 8.5 shows the algorithm automaton. Figure 8.6 shows the composite node automaton. The complete expanded and scheduled node automaton appears in Appendix A.3.

The formal parameters of each node are its identifier, the size of the network and a set specifying the neighbors with which the node may communicate. Note, that in the composite node automaton shown in Figure 8.6, each node is connected to every node in the network (including itself). This clique structure contrasts with the ring structure defined for `LCRNode` in Figure 3.7. In this design, the formal parameters to the algorithm automaton specify the network topology, and the algorithm automaton (and scheduled node automaton) respect that specification by exchanging messages only with nodes in its neighbors set. This design has the advantage that network topology can be specified

```
[[[[ transition: output Iprobe(4, 3) in automaton LCRNode(4)
 on loon.csail.mit.edu at 16:13:56:224
%%%% Modified state variables:
     RM --> Map, modified entries: {[3 -> Tuple, modified fields:
{[ready -> true] }]}
]]]]
[[[[ transition: output receive(4, 3) in automaton LCRNode(4)
 on loon.csail.mit.edu at 16:13:56:244
%%%% Modified state variables:
     RM --> Map, modified entries: {[3 -> Tuple, modified fields:
{[toRecv -> Sequence, elements added: {9 }  Elements removed: {}]
[ready -> false] }]}
]]]]
[[[[ transition: internal RECEIVE(9, 3, 4) in automaton LCRNode(4)
 on loon.csail.mit.edu at 16:13:56:246
%%%% Modified state variables:
     P --> Tuple, modified fields: {[pending -> ((9 1))] }
     RM --> Map, modified entries: {[3 -> Tuple, modified fields:
{[toRecv -> Sequence, elements added: {} Elements removed: {9 }] }]}
]]]]
[[[[ transition: internal SEND(9, 4, 5) in automaton LCRNode(4)
 on loon.csail.mit.edu at 16:13:56:248
%%%% Modified state variables:
     P --> Tuple, modified fields: {[pending -> ()] }
     SM --> Map, modified entries: {[5 -> Tuple, modified fields:
{[toSend -> Sequence, elements added: {9 } Elements removed: {}] }]}
]]]]
[[[[ transition: output Isend(9, 4, 5) in automaton LCRNode(4)
 on loon.csail.mit.edu at 16:13:56:252
%%%% Modified state variables:
     SM --> Map, modified entries: {[5 -> Tuple, modified fields:
{[toSend -> Sequence, elements added: {} Elements removed: {9 }]
[sent -> Sequence, elements added: {9 } Elements removed: {}]
[handles -> Sequence, elements added: {mpi.Request@df503 }
Elements removed: {}] }]}
]]]]
[[[[ transition: output test(mpi.Request@df503, 4, 5) in automaton LCRNode(4)
 on loon.csail.mit.edu at 16:13:56:256
%%%% Modified state variables:
     SM --> Map, modified entries: {[5 -> Tuple, modified fields:
{[handles -> Sequence, elements added: {} Elements removed:
{mpi.Request@df503 }] }]}
]]]]
```

Figure 8.4: An excerpt of example run of LCR leader election

```
type Message = enumeration of  search, nil

automaton spanProcess(i, size: Int, neighbors: Set[Int])
  signature
    input search (const i)
    input RECEIVE(m: Message, const i, j: Int)
    output SEND(m: Message, const i, j: Int)
    output parent(const i, j: Int)

  states
    searching: Bool := false,
    reported: Bool := false,
    parent: Int := -1,
    send: Map[Int, Message],
    nbrs: Set[Int],
    nbr: Int
    initially
        ∀ k:Int (defined(send, k) ⇔ k ∈ neighbors) ∧
        ∀ k:Int ((defined(send, k) ∧ send[k] = search) ⇔ i = 0)

  transitions
    input search(i)
      eff searching := true;
    input RECEIVE(m, i, j) where i = 0
    input RECEIVE(m, i, j) where i ≠ 0
      eff if parent = -1 then
            parent := j;
            for k: Int in neighbors - {j} do
              send[k] := search
            od
          fi
    output SEND(m, i, j)
      pre searching;
          send[j] = search;
          m = search
      eff send[j] := nil
    output parent(i, j) where i ≠ 0
      pre parent = j;
          parent ≥ 0;
          ¬reported;
          searching
      eff reported := true
    output parent(i, j) where i = 0
      pre ¬reported;
          searching
      eff reported := true
```

Figure 8.5: Algorithm automaton `spanProcess` specifies a participating process in an algorithm that constructs a spanning tree of an arbitrary connected network.

at runtime. For example, the neighbor relation might be built such that only nodes with relatively faster connections are neighbors.

The state of each node includes two boolean flags `searching` and `reported` which specify whether the node has begun actively participating in the search for the spanning tree and whether the node has announced its parent, respectively. In addition the state includes a `send` map that stores the next message to be sent to each neighbor and a variable specifying the parent of the node, if known. Initially, the flags are false and `parent` is set to -1 to indicate that the node's parent is unknown. The `send` map is empty at every node except the distinguished root node. At node 0, the `send` map is initialized such that node sends a search message to each of its neighbors. Finally, the state includes two auxiliary variables `nbrs` and `nbr` that are used in scheduling the node. They appear in the algorithm node only because the current prototype does not allow **initially det** blocks to declare their own variables or reference schedule variables.

The signature of the automaton has two console and two network actions. The console actions are the input `search` action, which toggles the `searching` flag to activate the node, and the output `parent` action, which announces the node's parent and toggles the `reported` flag. The network actions are the usual `SEND` and `RECEIVE` actions. When the node is active, the former sends a message stored in the `send` map and removes that message from the map. The first message received causes the node to set the sending node as its parent and to prepare to flood its neighbors with search messages. (The flood is only sent when the node is active.)

```
axioms Infinite(Handle)
axioms ChoiceSet(Int)
automaton spanNode(MPIrank, MPIsize: Int, neighbors: Set[Int])
  components
    P: spanProcess(MPIrank, MPIsize, neighbors);
    RM[j: Int]: ReceiveMediator(Message, Int, MPIrank, j);
    SM[j: Int]: SendMediator(Message, Int, MPIrank, j);
    I: spanProcessInterface(MPIrank, MPIsize, neighbors)
    hidden SEND(m, i, j), RECEIVE(m, j, i), parent(i, j), search(i)
```

Figure 8.6: Composite node automaton `spanNode` specifies one node in the spanning tree system.

8.3.1 Results

In our testbed, due to the broadcast nature of the interconnect, there are no marked disparities in connection times between nodes. We ran some initial experiments in which the neighbor relation specified that all nodes were neighbors. The resulting spanning tree was always a star with every node announcing the root as its parent. To avoid this rather uninteresting result, we specified a wrap-around mesh neighbor relation. An example twenty-node mesh is shown schematically in Figure 8.7.

All spanning tree experiments were run on a twenty-node mesh running on a network of ten

SAMPLES	MESSAGES	TRANSITIONS	IPROBES	RUNTIME (s)
54	61	40.1	61	22.7 (9.5)

Table 8.2: Measurements of the spanning tree algorithm on 20 nodes run on 10 machines. The columns are the number of the number of sample runs measured, the total number of messages sent across all channels in a run, the maximum number of transitions executed at any node in a run, the maximum number of `Isend`s performed at any node in a run, and the maximum runtime at any node on a run. Figures shown in the last four columns are averages across all sample runs. The standard deviation of the maximum runtime is shown in parenthesis. Runtimes times are measured in seconds.

machines. (Two nodes were allocated per machine). We ran the spanning tree algorithm fifty times. The trace of the output actions of a typical run is shown in Figure 8.8. The tree constructed is represented schematically in Figure 8.9. Since the algorithm simply floods the network with search messages, the number of messages sent by the algorithm is totally determined by the neighbor relation. One message is sent across every (directed) edge in the network. In a wrap-around mesh network every node has degree four. Thus, each run of the algorithm sent 80 messages. In our measurements, we take the maximum runtime at any node as the runtime of the algorithm. As with LCR, the number of transitions executed is dominated by the number of **Iprobe** calls performed. In fact, unless a **test** call returns false (a phenomenon not observed), all variability in the number of transitions executed is due to the number of **Iprobe** calls executed. Table 8.2 summarizes our measurements.

8.4 Asynchronous Broadcast/Convergecast

The third algorithm we use is an extension of the $AsynchBcastAck_i$ algorithm presented in Section 15.3 of [81] and of an earlier IOA implementation of that algorithm by Mavrommatis and Georgiou [88]. This version of the algorithm combines three phases. First, it builds a spanning tree for the network. This phase is just a slightly different implementation of the previous algorithm. Second, the root broadcasts messages over the spanning tree. Finally, when the root broadcasts a distinguished "last" value, the leaves convergecast an acknowledgment back to the root.

Initially, all nodes in the network are idle. Inputs are only processed at a distinguished root node (in this case, node 0). The formal parameters of each node are its set of neighbors and the distinguished last value. Console inputs consist of messages to be broadcast. Console inputs at any node except the root are ignored.

When the root is awakened by the first input action, it begins a phase to build a spanning tree on the network. The root floods its neighbors with wakeup messages. Upon receipt of a wakeup message, each node selects the sending node as its parent in the spanning tree and proceeds to flood its neighbors. If a node receives a wakeup message after it has selected a parent (or if it is the parent-

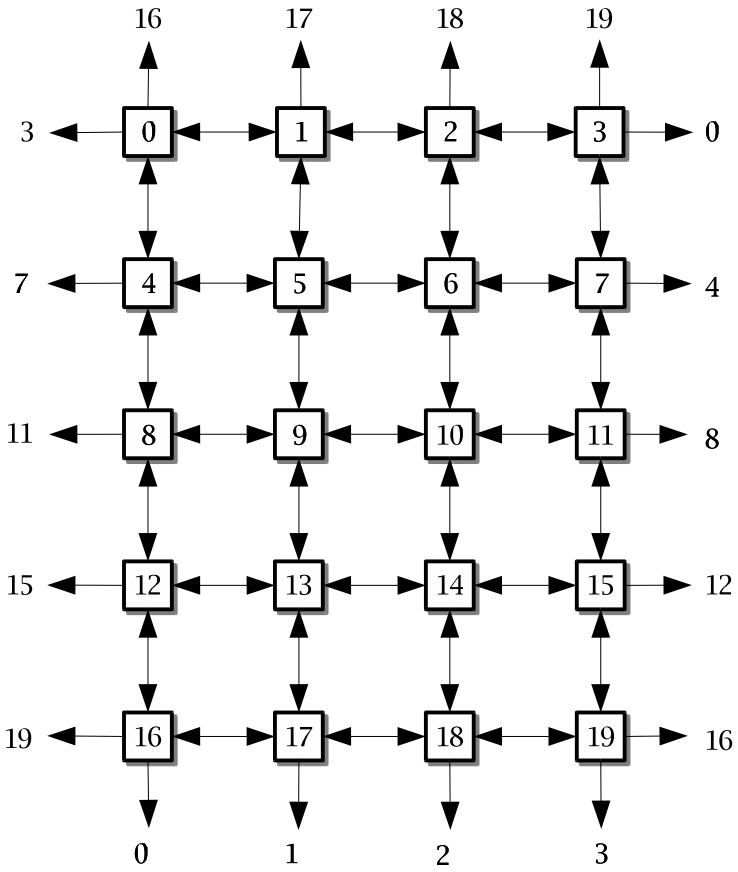

Figure 8.7: A 4 x 5 node wrap-around mesh network

less root node), it immediately responds with a negative acknowledgment (nack). When a node receives a positive acknowledgment (ack), it adds the acknowledging node to its set of children. A node announces its parent with the `parent` output action and sends a positive acknowledgment to its parent after it has received acknowledgments (negative or positive) from all its neighbors other than its parent. Since only the leaves of the spanning tree initially receive (negative) acknowledgments from all their neighbors, the positive acknowledgment messages are convergecast up the tree from the leaves. When the root has received acknowledgments from all its children, the spanning tree is complete and broadcasts may begin.

Since broadcast requests may arrive at the root at any time, it buffers them until the spanning tree has been set up. After the tree has been initialized and each node has announced its parent, an internal action begins copying messages from the buffer onto individual queues of messages destined for each child of the root. Each intermediate node in the spanning tree forwards any message it receives to its children and reports the message as received in an output action.

When the special "last" message is broadcast a second convergecast begins. This time each node waits until it has reported all broadcasts and for acks from its children before sending an

```
[action: parent, params: {Int(14), Int(15)}]
[action: parent, params: {Int(17), Int(0)}]
[action: parent, params: {Int(7), Int(3)}]
[action: parent, params: {Int(16), Int(0)}]
[action: parent, params: {Int(9), Int(5)}]
[action: parent, params: {Int(3), Int(0)}]
[action: parent, params: {Int(2), Int(1)}]
[action: parent, params: {Int(18), Int(19)}]
[action: parent, params: {Int(12), Int(16)}]
[action: parent, params: {Int(19), Int(3)}]
[action: parent, params: {Int(10), Int(6)}]
[action: parent, params: {Int(1), Int(0)}]
[action: parent, params: {Int(6), Int(5)}]
[action: parent, params: {Int(4), Int(0)}]
[action: parent, params: {Int(15), Int(19)}]
[action: parent, params: {Int(13), Int(9)}]
[action: parent, params: {Int(8), Int(4)}]
[action: parent, params: {Int(5), Int(4)}]
[action: parent, params: {Int(0), Int(-1)}]
[action: parent, params: {Int(11), Int(7)}]
```

Figure 8.8: Typical output of the spanning tree algorithm on a 4 x 5 node wrap-around mesh network

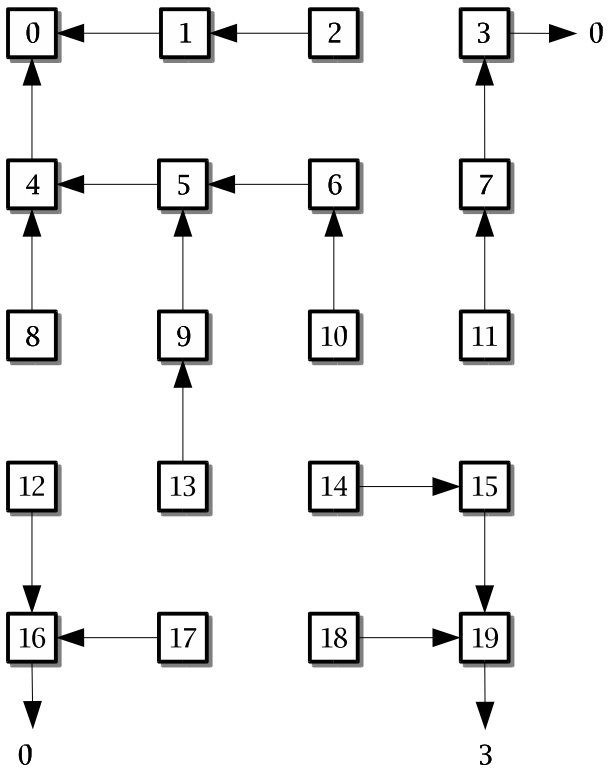

Figure 8.9: Typical spanning tree computed on a 4 x 5 node wrap-around mesh network

acknowledgment to its parent. Again since leaves have no children, the acks propagate up the tree. The schedule of each node may terminate after it has sent this final acknowledgment. When the root node has received acks from all its children, it and the entire algorithm may terminate. Notice that since the root node is the first to awake and the last to complete, the runtime of the entire algorithm may be determined at the root node alone.

The number of messages sent by this algorithm is totally determined by the number of nodes and number of edges in the network. Setting up the spanning tree requires two messages per edge in the network: one search message and one ack or nack. Sending a broadcast requires one message per edge in the spanning tree. To convergecast an acknowledgment of the last broadcast requires an additional message per edge in the spanning tree.

In our experiment, we use a wrap-around mesh network with R rows and C columns and $N = R \times C$ nodes. During spanning tree initialization each node sends a search message to and receives an acknowledgment from each of its neighbors except its parent. Each node in a wrap-around mesh has degree four. Thus, the root node sends four search messages and four nacks and every other node sends three search messages and three acknowledgments (positive or negative). Thus, setting up the spanning tree in a wrap-around mesh requires $6N + 2$ messages. There are, by definition, $N - 1$ edges in a spanning tree. Therefore, sending a broadcast or a convergecast requires $N - 1$ messages. Thus to send M messages plus the acknowledging final convergecast in an N node wrap-around mesh network requires $(6N + 2) + (M + 1)(N - 1)$ messages. For example to send 100 messages in a 3×3 network requires $56 + 808 = 864$ total messages.

8.4.1 Results

In our broadcast experiment, we again measured the number of messages sent, the number of transitions executed, the number of **Iprobe** calls executed, and the algorithm runtime. In this experiment we varied the number of broadcasts sent through a wrap-around mesh network of nine nodes from eleven to 751. Our results are summarized in Table 8.3. Average runtime is plotted against the number of broadcasts in Figure 8.13. Average number of transitions and **Iprobe**s executed and the measured total number messages per run are plotted against the same abscissa in Figure 8.14.

The runtime of the algorithm and the number of transitions and **Iprobe**s executed grow quadratically with the number of broadcasts. The result is quite unexpected because, as discussed above, the number of messages sent grows only linearly with the number of broadcasts. In fact, the message counts conform exactly to formula calculated above. Compilation adds no overhead in number of messages sent.

We can attribute this quadratic growth to our naïve implementation of datatypes. The standard library implements every instance of a datatype as an immutable object. Even collections such as

```
axioms Null(Int)
type Status = enumeration of idle, initializing, announced,
                             bcasting, finalizing, done
type Kind = enumeration of bcast, ack, nack
type Message = tuple of kind: Kind, payload: Null[Int]

automaton bcastProcess(i, size: Int, neighbors: Set[Int], last: Int)
  signature
    input bcast(const i, v: Int)
    internal queue(const i, v:Int)
    internal ackInit(const i)
    internal ackLast(const i)
    input RECEIVE(m: Message, const i, j: Int)
    output SEND(m: Message, const i, j: Int)
    output parent(const i, j: Null[Int])
    output report(const i, v: Int)

  states
    status: Status := idle,
    parent: Null[Int] := nil,
    children: Set[Int] := {},
    acked: Set[Int] := {},
    outgoing: Seq[Int] := {},
    incomming: Seq[Int] := {},
    send: Map[Int, Seq[Message]],
    nbrs: Set[Int],
    nbr: Int
    initially
        ∀ k:Int (defined(send, k) ⇔ k ∈ neighbors) ∧
        ∀ k:Int (i = 0 ⇒ (defined(send, k) ∧ head(send[k]) = [bcast, nil]))

  transitions
    input bcast(i, v) where i ≠ 0
    input bcast(i, v) where i = 0
      eff if status = idle then
             status := initializing;
             for k: Int in neighbors do
               send[k] := send[k] ⊢ [bcast, nil]
             od
          fi;
          outgoing := outgoing ⊢ v;

    internal queue(i, v) where i ≠ 0
    internal queue(i, v) where i = 0
      pre status = bcasting;
          outgoing ≠ {};
          v = head(outgoing)
      eff outgoing := tail(outgoing);
          for k: Int in children do
              send[k] := send[k] ⊢ [bcast, embed(v)];
          od;
          if v = last then
             status := finalizing;
             acked := {}
          fi
```

Figure 8.10: Beginning of algorithm automaton `bcastProcess` specifies a participating process in an algorithm that constructs a spanning tree and performs repeated broadcasts along that tree until a distinguished `last` value is broadcast.

```
internal ackInit(i)
  pre status = announced
  eff status := bcasting;
      if i ≠ 0 then
         send[parent.val] := send[parent.val] ⊢  [ack, nil]
      fi

internal ackLast(i)
  pre status = finalizing;
      acked = children;
      incomming = {}
  eff status := done;
      if i ≠ 0 then
         send[parent.val] := send[parent.val] ⊢  [ack, nil]
      fi

input RECEIVE(m, i, j) where m.kind = nack
  eff acked := acked ∪ {j};
input RECEIVE(m, i, j) where m.kind = ack
  eff acked := acked ∪ {j};
      if status = initializing then children := children ∪ {j}; fi
input RECEIVE(m, i, j) where m.kind = bcast ∧ m.payload = nil
  eff if parent = nil ∧ i ≠ 0 then
         parent := embed(j);
         status := initializing;
         acked := {parent.val};
         for k: Int in neighbors - {parent.val} do
             send[k] := send[k] ⊢ m
         od
      else
         send[j] := send[j] ⊢ [nack, nil]
      fi
input RECEIVE(m, i, j) where m.kind = bcast ∧ m.payload ≠ nil
  eff incomming := incomming ⊢ m.payload.val;
      for k: Int in children do
          send[k] := send[k] ⊢ m
      od;
      if m.payload.val = last then
         status := finalizing;
         acked := {};
      fi

output SEND(m, i, j)
  pre send[j] ≠ {};
      m = head(send[j])
  eff send[j] := tail(send[j])

output parent(i, j)
  pre status = initializing;
      acked = neighbors;
      parent = j
  eff status := announced;

output report(i, v)
  pre  incomming ≠ {};
       v = head(incomming)
  eff  incomming := tail(incomming)
```

Figure 8.11: Remainder of algorithm automaton `bcastProcess` specifies a participating process in an algorithm that constructs a spanning tree and performs repeated broadcasts along that tree until a distinguished `last` value is broadcast.

```
axioms Infinite(Handle)
axioms ChoiceSet(Int)
automaton bcastNode(MPIrank, MPIsize: Int, neighbors: Set[Int], last: Int)
  components
    P: bcastProcess(MPIrank, MPIsize, neighbors, last);
    RM[j: Int]: ReceiveMediator(Message, Int, MPIrank, j);
    SM[j: Int]: SendMediator(Message, Int, MPIrank, j);
    I: bcastProcessInterface(MPIrank, MPIsize, neighbors, last)
    hidden SEND(m, i, j), RECEIVE(m, i, j),
           bcast(i, v), parent(i, j), report(i,v)
```

Figure 8.12: Composite node automaton `bcastNode` specifies one node in the broadcast system.

BROADCASTS	SAMPLES	MESSAGES	TRANSITIONS	IPROBES	RUNTIME (σ)	
11	67	152	554	389	8.41	(4.3)
21	98	232	828	537	11.9	(4.5)
31	98	312	1140	721	16.8	(6.4)
41	72	392	1330	780	18.8	(5.9)
51	97	472	1570	893	21.7	(6.5)
101	99	872	2600	1420	38.3	(8.1)
151	97	1272	3780	2610	62.4	(10)
201	14	1672	7330	6140	132	(39)
251	95	2072	8510	7020	150	(23)
301	96	2472	12100	10700	227	(30)
351	20	2872	17400	16000	340	(28)
401	20	3272	29600	28000	594	(160)
451	20	3672	41600	40200	840	(220)
501	20	4072	52400	50900	1070	(230)
751	19	6072	177000	170000	3560	(530)

Table 8.3: Measurements of the broadcast algorithm. The columns are the number of broadcasts sent through the spanning tree, the number of sample runs measured, the total number of messages sent across all channels in a run, the maximum number of transitions executed at any node in a run, the maximum number of `Isend`s performed at any node in a run, and the maximum runtime at any node on a run. Figures shown in the last four columns are averages across all sample runs. The standard deviation of the maximum runtimes is shown in parenthesis. Runtimes times are measured in seconds.

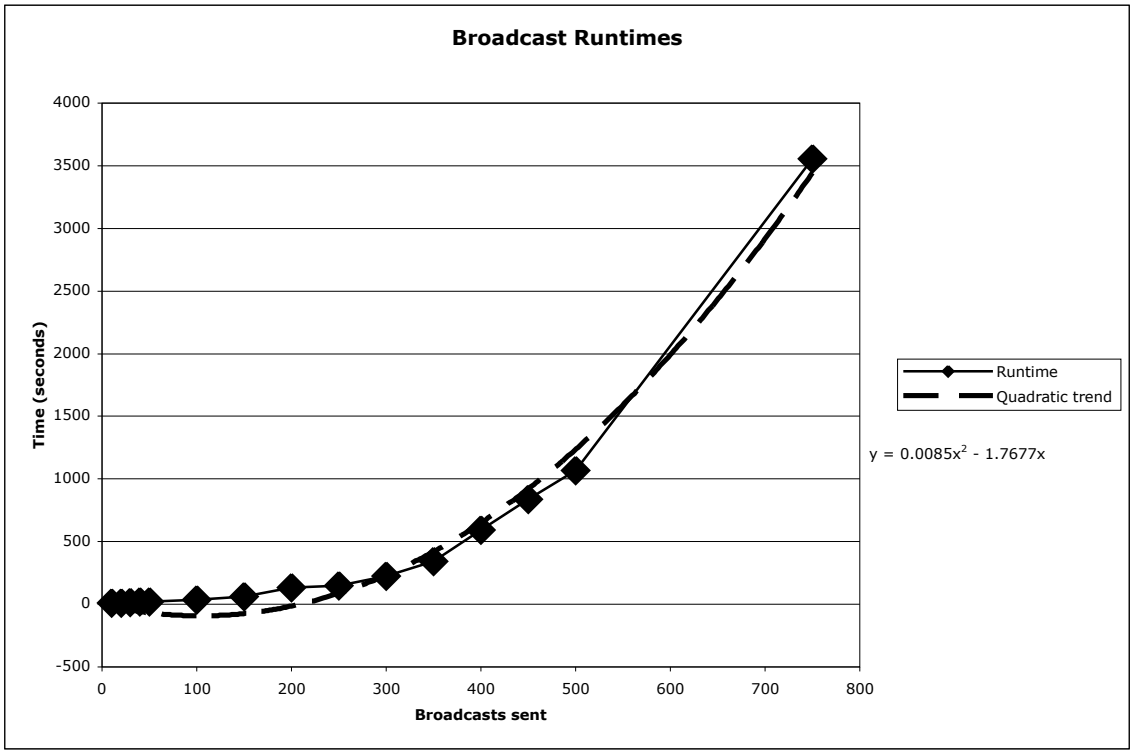

Figure 8.13: Broadcast runtimes

arrays, maps, sets, and sequences are implemented this way. Thus, in the current implementation every change to a variable that stores a collection entails copying all the unchanged parts of the collection to generate the new instance. For example, in our current implementation appending an element to a sequence causes the entire sequence to be copied. As a result, sequence append takes time proportional to the length of the sequence. Thus, a sequence of appends (without any intervening operations) takes time proportional to the square of the number of appends.

8.5 Observations

Programming algorithms from the literature with IOA was generally a smooth process. An undergraduate student with only minimal experience with IOA and I/O automata was able to implement versions of the asynchronous spanning tree and broadcast/convergecast algorithms in a matter of hours. Writing schedules was both easier and harder than expected: For the algorithms in our case studies, schedules followed fairly predictable patterns. The arrival of a message or an input action triggers the execution of a cascade of transitions. The schedules for our case studies essentially loop over these possible sources of input and when an input arrives the schedule performs the entire resulting cascade of transitions before checking for the next input. Thus, the basic structure of a schedule turned out to be very easy to outline. On the other hand, our experience was that most

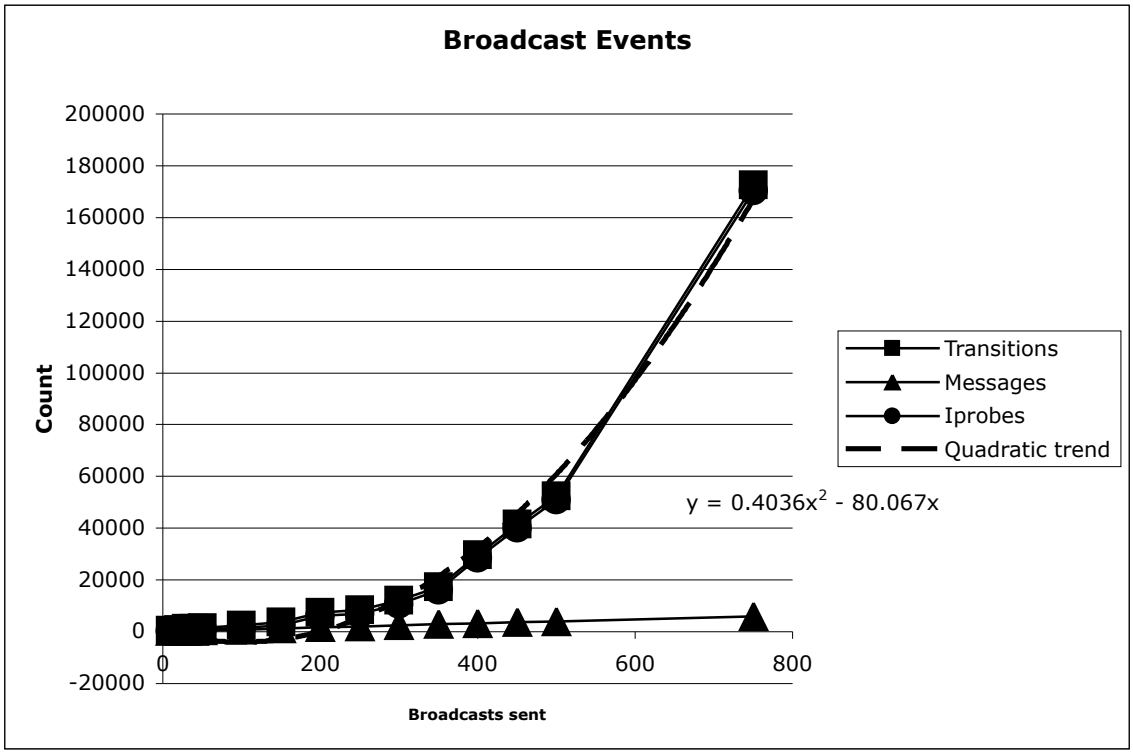

Figure 8.14: Broadcast counts

programming time was spent debugging NDR schedules. In this regard, runtime checks on NDR generated values (*e.g.*, precondition checks) proved valuable. Unfortunately, livelock was an all too frequent result of a buggy schedule. Writing the conditional guards for **fire** statements was particularly tricky when polling for items from the `stdin` queue. In particular, it was a frequent bug that a schedule never executed any transitions.

Part II

IOA Composer

Chapter 9

Introduction

"Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," the King said, gravely, "and go on till you come to the end: then stop."

— Lewis Carroll [16]

In our IOA compilation strategy, the programmer composes a basic algorithm automaton with a number of channel mediator and console buffer automata to create a composite node automaton. However, the IOA compiler translates only primitive IOA programs into Java. Therefore, the composite automaton must be expanded into primitive form before compilation. We have designed and implemented a *composer* tool to translate a composite IOA automaton into a semantically equivalent primitive automaton. This part describes, both formally and with examples, the constraints on the definitions of primitive and composite IOA automata, the composability requirements for the components of a composite automaton, and the syntactic transformation of a composite automaton into an equivalent primitive automaton.

Chapter 10 introduces four examples used throughout this part to illustrate new definitions and operations. Chapter 11 treats IOA programs for primitive I/O automata: it introduces notations for describing the syntactic structures that appear in these programs, and it lists syntactic and semantic conditions that these programs must satisfy to represent valid primitive I/O automata. Chapter 12 describes how to reformulate primitive IOA programs into an equivalent but more regular (desugared) form that is used in later definitions in this part. Chapter 13 treats IOA programs for composite I/O automata: it introduces notations for describing the syntactic structures that appear in these programs, describes resortings induced by them, and lists syntactic and semantic conditions that these programs must satisfy to represent valid composite I/O automata. Chapter 14 describes the translation of the name spaces of component automata into a unified name space

for the composite automaton. Chapter 15 shows how to expand an IOA program for a composite automaton into an equivalent IOA program for a primitive automaton. The expansion is generated by combining syntactic structures of the desugared programs for the component automata after applying appropriate replacements of sorts and variables. Chapter 16 details the expansion of the composite automaton introduced in Chapter 10 using the desugared forms developed throughout Chapters 12–14 and the techniques described in Chapter 15. Finally, Chapter 17 gives a precise definition of the resortings and substitutions used to replace sorts and variables.

This part is also published as MIT LCS Technical Report 959 co-authored with Steve Garland [118].

Chapter 10

Illustrative examples

And is then example nothing? It is every thing. Example is the school of mankind, and they will learn at no other.

— Edmund Burke [15]

We use several examples of primitive and composite automata to illustrate both the notations provided by IOA and also the formal semantics of IOA. We refer to Examples 10.1–10.3 throughout Sections 11–16. Example 10.4 is relevant only to Sections 13–16.

Example 10.1 Figure 10.1 contains an IOA specification for a communication channel that can both drop duplicate messages and reorder messages. Type parameters for the specification, `Node` and `Msg`, represent the set of nodes that can be connected by channels and the set of messages that can be transmitted. Individual parameters, `i` and `j`, represent the nodes connected by a particular channel.

Two features of this example warrant particular attention later in this Part. First, the example uses both type and variable automaton parameters. Second, it uses the keyword **const** to indicate that the parameters `i` and `j` in the action signature are terms referring to the parameters `i` and `j` of the automaton, rather than fresh variable declarations.

Example 10.2 Figure 10.2 contains the specification for a process that runs on a node indexed by a natural number and that communicates with its neighbors by sending and receiving messages that consist of natural numbers. The process records the smallest value it has received and passes on all values that exceed the recorded value; if the set of values waiting to be passed on grows too large, the process can also lose a nondeterministic set of those values. Interesting features of this example

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(const i, const j, m:Msg)
    output receive(const i, const j, m:Msg)
  states contents:Set[Msg] := {}
  transitions
    input send(i, j, m)
      eff contents := insert(m, contents)
    output receive(i, j, m)
      pre m ∈ contents
      eff contents := delete(m, contents)
```

Figure 10.1: Sample automaton `Channel`

```
automaton P(n:Int)
  signature
    input receive(const n-1, const n, x:Int)
    output send(const n, const n+1, x:Int),
           overflow(const n, s:Set[Int])
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(n-1, n, x)
      eff if val = 0 then val := x
          elseif x < val then
            toSend := insert(val, toSend);
            val := x
          elseif val < x then
            toSend := insert(x, toSend)
          fi
    output send(n, n+1, x)
      pre x ∈ toSend
      eff toSend := delete(x, toSend)
    output overflow(n, s:Set[Int]; local t:Set[Int])
      pre s = toSend ∧ n < size(s) ∧ t ⊆ s
      eff toSend := t
```

Figure 10.2: Sample automaton `P`

include the use of terms as parameters in transition definitions and a **local** variable representing an initial nondeterministic choice and temporary state local to the transition. (The keyword **local**, newly added to the IOA language, replaces and extends the keyword **choose** formerly used to introduce hidden parameters. See Chapter 11 for a fuller description of **local** parameters.)

Example 10.3 Figure 10.3 contains the specification for another process that watches for `overflow` actions and reports those that meet a simple criterion. Interesting features of this example include more complicated uses of type parameters and **where** clauses, both in the action signature and to distinguish two transition definitions for a single action.

Example 10.4 Finally, Figure 10.4 contains the specification of an automaton formed by composing

```
automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s ∪ {x})
      eff seen[x] := true
    input overflow(x, s) where ¬(x ∈ s)
      eff seen[x] := false
    output found(x)
      pre seen[x]
```

Figure 10.3: Sample automaton `Watch`

```
axioms Between(Int, ≤)

automaton Sys(nProcesses: Int)
  components C[n:Int]: Channel(Int, Int, n, n+1)
                  where 1 ≤ n ∧ n < nProcesses;
             P[n:Int] where 1 ≤ n ∧ n ≤ nProcesses;
             W: Watch(Int, between(1, nProcesses))
  hidden send(nProcesses, nProcesses+1, m)

invariant of Sys:
  ∀ m:Int ∀ n:Int (1 ≤ m ∧ m < n ∧ n ≤ nProcesses
                        ⇒ P[m].val < P[n].val ∨  P[n].val = 0)
```

Figure 10.4: Sample composite automaton `Sys`

instances of these three primitive automata. This specification relies on an auxiliary specification, shown in Figure 10.5, to define the term `between(1, nProcesses)`.

```
Between(T, ≤:T,T→Bool): trait
  includes Set(T)
  introduces
    __≤__: T, T → Bool
    between: T, T → Set[T]
  asserts with x, y, z: T
    x ∈ between(y, z) ⇔ y ≤ x ∧ x ≤ z
```

Figure 10.5: Auxiliary definition of function `between`

Chapter 11

Definitions for primitive automata

The beginning of wisdom is calling things by their right names.

– Chinese Proverb

In order to describe syntactic manipulations of IOA programs, we introduce a nomenclature for their syntactic elements. We expose just those elements of an IOA program we use to describe the expansion of composite automata into primitive form. Section 11.1 introduces nomenclature for, and the meaning of, syntactic structures in primitive automata. Section 11.2 examines how states are represented and referenced in primitive IOA programs. Sections 11.3 and 11.4 describe semantic conditions that must hold for an IOA program to represent a valid primitive I/O automaton.

11.1 Syntax

Figure 11.1 illustrates the general form of an IOA definition for a primitive I/O automaton. The figure exposes just those elements of an IOA program we use to describe the expansion of composite automata into primitive form. It does not expose the individual statements that appear in an **eff** clause. (These are treated separately in Chapter 17.) Rather the figure simply refers to the "program" (*i.e.*, the complete sequence of statements) in an **eff** clause.

11.1.1 Notations and writing conventions

In Figure 11.1, $params^A$ denotes the sequence of type and variable declarations that serve as the parameters of the automaton A. The *Assumptions* are LSL theories defining required properties for these parameters. Notations $params^{A,\pi}_{kind}$ and $params^{A,\pi}_{kind,t_j}$, where *kind* is one of *in*, *out*, or

automaton $A(params^A)$

assumes $Assumptions$

signature

...

input $\pi(params_{in}^{A,\pi})$ **where** $P_{in}^{A,\pi}$

output $\pi(params_{out}^{A,\pi})$ **where** $P_{out}^{A,\pi}$

internal $\pi(params_{int}^{A,\pi})$ **where** $P_{int}^{A,\pi}$

...

states $stateVars^A := initVals^A$ **initially** P_{init}^A

transitions

...

input $\pi(params_{in,t_j}^{A,\pi};$ **local** $localVars_{in,t_j}^{A,\pi})$ **case** t_j **where** $P_{in,t_j}^{A,\pi}$

eff $Prog_{in,t_j}^{A,\pi}$ **ensuring** $ensuring_{in,t_j}^{A,\pi}$

output $\pi(params_{out,t_j}^{A,\pi};$ **local** $localVars_{out,t_j}^{A,\pi})$ **case** t_j **where** $P_{out,t_j}^{A,\pi}$

pre $Pre_{out,t_j}^{A,\pi}$

eff $Prog_{out,t_j}^{A,\pi}$ **ensuring** $ensuring_{out,t_j}^{A,\pi}$

internal $\pi(params_{int,t_j}^{A,\pi};$ **local** $localVars_{int,t_j}^{A,\pi})$ **case** t_j **where** $P_{int,t_j}^{A,\pi}$

pre $Pre_{int,t_j}^{A,\pi}$

eff $Prog_{int,t_j}^{A,\pi}$ **ensuring** $ensuring_{int,t_j}^{A,\pi}$

...

Figure 11.1: General form of a primitive automaton

int, denote sequences of variables and/or terms that serve as parameters for the action π and its transition definitions. The notations $P_{kind}^{A,\pi}$, P_{init}^{A}, $P_{kind,t_j}^{A,\pi}$, $Pre_{kind,t_j}^{A,\pi}$, and $ensuring_{kind,t_j}^{A,\pi}$ denote predicates (*i.e.*, boolean-valued expressions). The notation $initVals^A$ denotes the sequence of terms or **choose** expressions serving as initial values for the state variables. If the definition of A does not specify an initial value for some state variable, we treat the declaration of that state variable as equivalent to one of the form `x:T :=` **choose** `t:T` **where** `true`. The notation $Prog_{kind,t_j}^{A,\pi}$ denotes a program. The notation $localVars_{kind,t_j}^{A,\pi}$ denotes a sequence of variables. In general, a notation ending with an "s" denotes a sequence of zero or more elements.

Our conventions for decorating syntactic structures throughout this paper are as follows. Superscripts refer either to automaton names or to automaton-name/action-name pairs. Automaton names are capitalized (*e.g.*, A, C_i, `P`). Action names are not capitalized and are either Greek letters (*e.g.*, π, π_1) or written in `mono-spaced font` (*e.g.*, `send`). Subscripts refer to more specific restrictions such as action kind (*i.e.*, *in*, *out*, or *int*), transition label (*e.g.*, t_1), or origin (*e.g.*, *desug*). IOA keywords appear in a **small-bold roman font**. References to other text in sample IOA programs appear in a `mono-spaced font`. Syntactic structure labels and names in general IOA programs are

italicized.

11.1.2 Syntactic elements of primitive IOA programs

Variables in IOA programs can be declared explicitly as automaton parameters ($vars^A$, which is a subsequence of $params^A$), as state variables ($stateVars^A$), or as local variables ($localVars^{A,\pi}_{kind,t_j}$); they can also be declared implicitly as post-state variables that correspond to state variables, post-local variables corresponding to local variables, or by their appearance in action parameters ($vars^{A,\pi}_{in}$, which appear in $params^{A,\pi}_{in}$) or in transition parameters ($vars^{A,\pi}_{in,t_j}$, which appear in $params^{A,\pi}_{in,t_j}$). Variables in IOA programs can appear in parameters, terms, predicates, and programs. For simplicity, Figure 11.1 does not indicate which variables may have free occurrences in which parameters, terms, predicates, or programs; Section 11.3 describes which can occur where. As an illustration, variables that occur freely in $P^{A,\pi}_{in}$ must be in one of the sequences $vars^A$ or $vars^{A,\pi}_{in}$.

Below, we define each labeled syntactic structure and then illustrate it using selections from Examples 10.1–10.3.

11.1.3 Parameters

- $params^A$ is the sequence of formal parameters for A, which can be either variables or **type** parameters. We decompose $params^A$ into two disjoint subsequences, one ($vars^A$) containing variable declarations and the other ($types^A$) containing **type** parameters (identifiers qualified by the keyword **type**). For example, $params^{\texttt{Watch}}$ is ⟨`T:`**`type`**`, what:Set[T]`⟩, which consists of a **type** parameter `T` followed by a variable `what:Set[T]`. Hence $types^{\texttt{Watch}}$ is ⟨`T:`**`type`**⟩ and $vars^{\texttt{Watch}}$ is ⟨`what:Set[T]`⟩.

- $params^{A,\pi}_{kind}$ is the sequence of parameters for the set of actions of type *kind* named by π in A's signature. Action parameters can be either variables or **const** terms.[1] For example, $params^{\texttt{Channel},\texttt{send}}_{in}$ is ⟨**`const i, const j, m:Msg`**⟩.

- $params^{A,\pi}_{kind,t_j}$ is the sequence of terms serving as parameters for transition definition t_j for actions of type *kind* named by π. Whereas π can appear at most once as the name of an input, output, and internal action in A's signature, it can have more than one transition definition as an input, output, and internal action. For example, $params^{\texttt{Watch},\texttt{overflow}}_{in,t_1}$ is ⟨`x, s ∪ {x}`⟩ and $params^{\texttt{Watch},\texttt{overflow}}_{in,t_2}$ is ⟨`x, s`⟩.

[1] We may want to consider an alternative treatment for action parameters, similar to that for $params^{A,\pi}_{kind,t_j}$, that would dispense with the keyword **const** and treat all action parameters as terms, rather than as a mixture of terms and variable declarations. The current treatment allows factored notations, such as $\pi(i, j{:}Int)$, which introduce a list of variables of a given sort; the alternative treatment would require unfactored notations, such as $\pi(i{:}Int, j{:}Int)$, in which a sort qualification applies only to the term it follows immediately.

11.1.4 Variables

- As noted above, $vars^A$ is the sequence of variables that are declared explicitly in $params^A$, that is, $vars^A$ is the sequence of identifiers in $params^A$ qualified by some sort other than **type**.[2] For example, $vars^{\texttt{Channel}}$ is ⟨`i:Node, j:Node`⟩.

- $vars^{A,\pi}_{kind}$ is the sequence of variable declarations (*i.e.*, non-**const** parameters) in $params^{A,\pi}_{kind}$. For example, $vars^{\texttt{Channel,send}}_{in}$ is ⟨`m:Msg`⟩.

- $stateVars^A$ is the sequence of state variables of A. For example, the sequence $stateVars^{\texttt{Channel}}$ is ⟨`contents:Set[Msg]`⟩.

- $postVars^A$ is the sequence of variables for post-states of A that can occur in any $ensuring^{A,\pi}_{kind,t_j}$. These variables are primed versions of variables in $stateVars^A$. For example, $postVars^{\texttt{P}}$ is ⟨`val′:Int, toSend′:Set[Int]`⟩.[3]

- $vars^{A,\pi}_{kind,t_j}$ is the sequence of variables that occur freely in $params^{A,\pi}_{kind,t_j}$, but are not in $vars^A$. For example, $vars^{\texttt{P,send}}_{out,t_1}$ is ⟨`x:Int`⟩, because `n` is in $vars^{\texttt{P}}$.

- $localVars^{A,\pi}_{kind,t_j}$ is a sequence of additional **local** variables for transition definition t_j for actions of type *kind* named π; these variables are not listed as parameters of π in the signature of A. For example, $localVars^{\texttt{P,overflow}}_{out,t_1}$ is ⟨`t:Set[Int]`⟩.

- $localPostVars^{A,\pi}_{kind,t_j}$ is the sequence of *post-local* variables that name the values of local variables after execution of $Prog^{A,\pi}_{kind,t_j}$. These variables are primed versions of variables in $localVars^{A,\pi}_{kind,t_j}$ that appear on the left side of an assignment statement in the transition definition and that can occur in $ensuring^{A,\pi}_{kind,t_j}$.

11.1.5 Predicates

- $P^{A,\pi}_{kind}$ is the **where** clause for the set of actions of type *kind* named by π in A's signature. For example, $P^{\texttt{Watch,found}}_{out}$ is `x ∈ what`. If $P^{A,\pi}_{kind}$ is not specified explicitly, it is taken to be *true*. If action π does not appear as a particular kind—input, output, or internal—in A's signature, then $P^{A,\pi}_{kind}$ is defined to be *false*.

[2] When we define a sequence by selecting some members of another sequence, we preserve order in projecting from the defining sequence to the defined sequence. For example, if $u{:}S$ precedes $v{:}T$ in $params^A$, then $u{:}S$ precedes $v{:}T$ in $vars^A$.

[3] Previously, only the primed versions of state variables that appeared on the left side of an assignment statement in the transition definition were allowed to appear in an **ensuring** clause. For example, we defined $postVars^{\texttt{P,send}}_{out,t_1}$ to be ⟨`toSend′:Set[Int]`⟩, which did not include the variable `val′`, because `val` does not appear on the left side of an assignment in this transition definition. The more complicated definition was intended as a safeguard against specifiers writing `val′` in an **ensuring** clause when there was no way the value of `val′` could differ from that of `val`. However, the more complicated definition did not safeguard against all such errors, because specifiers could still write `A′.val` in an **ensuring** clause. Hence the simpler definition appears preferable.

- P^{A}_{init} is a predicate constraining the initial values for A's state variables. If it is not specified explicitly, it is taken to be *true*.

- $P^{A,\pi}_{kind,t_j}$ is the **where** clause for transition definition t_j for actions of type *kind* named by π. For example, $P^{\texttt{Watch,overflow}}_{in,t_2}$ is $\neg$`(x ∈ s)`. If $P^{A,\pi}_{kind,t_j}$ is not specified explicitly, it is taken to be *true*. If action π does not appear as a particular kind in A's signature, then $P^{A,\pi}_{kind,t_j}$ is defined to be *false*.

- $Pre^{A,\pi}_{kind,t_j}$ is the precondition for transition definition t_j for actions of type *kind* named π, where *kind* is *out* or *int*. For example, $Pre^{\texttt{P,send}}_{out,t_1}$ is `x ∈ toSend`. If $Pre^{A,\pi}_{kind,t_j}$ is not specified explicitly, it is taken to be *true*. For every input transition, $Pre^{A,\pi}_{in,t_j}$ is defined to be *true* because transition definitions for input actions do not have preconditions.

- $ensuring^{A,\pi}_{kind,t_j}$ is the **ensuring** clause in the effects clause in transition definition t_j for actions of type *kind* named π. If $ensuring^{A,\pi}_{kind,t_j}$ is not specified explicitly, it is taken to be *true*. In the examples, all **ensuring** clauses are `true` by default.[4]

11.1.6 Programs and values

- $Prog^{A,\pi}_{kind,t_j}$ is the program in the effects clause in transition definition t_j for actions of type *kind* named π. For example, $Prog^{\texttt{P,overflow}}_{out,t_1}$ is `toSend := t`.

- $initVals^{A}$ is the sequence of initial values for A's state variables, which can be specified as either terms or **choose** expressions. A state variable without an explicit initial value is equivalent to one with an unconstrained initial value, that is, to one specified by a **choose** expression constrained by the predicate *true*. For example, $initVals^{\texttt{P}}$ is $\langle$`0, {}`$\rangle$.

- t_j is an optional identifier used to distinguish transition definitions of the same *kind* for the same action π. If there is no **case** clause, t_j is taken to be an arbitrary, but unique label.[5]

[4]The keyword **ensuring** replaces the **so that** keyword, which has been removed from IOA. Formerly, **so that** was used to introduce three types of predicates in IOA: the initialization predicate for automaton state, the post-state predicate for transition definitions, and the loop variable predicate in **for** statements. This multiple use was confusing. Furthermore, the keyword **where** also introduces predicates, which led to additional confusion. In the new syntax, automaton state predicates are introduced by **initially**, post-state predicates are introduced by **ensuring**, and all other predicates (including **for** predicates) are introduced by **where**. The semantics of the clauses containing these predicates has not changed.

[5]The **case** clause was introduced for use by the IOA simulator; it is not described yet in the IOA manual.

11.2 Aggregate sorts for state and local variables

11.2.1 State variables

The value (or the lvalue) of any state variable (*e.g.*, `toSend:Set[Int]`) may be referenced using that variable (*e.g.*, `toSend`) as if it were a constant operator (*e.g.*, `toSend: → Set[Int]`).[6] However, in contexts that involve more than a single automaton (*e.g.*, simulation relations or composite automata), such variable references may be ambiguous. Hence IOA provides an equivalent, unambiguous notation for the values of state variables.

For each automaton A without type parameters, IOA automatically defines a sort $States[A]$, known as the *aggregate state sort* of A, as a tuple sort with a selection operator $\_\_.v{:}States[A] \rightarrow T$ for each state variable v of sort T. IOA also automatically defines variables A and A' of sort $States[A]$ to represent the *aggregate state* and *aggregate post-state* of A. The terms $A.v$ and $A'.v$ are equivalent to references to the state variable v and to its value v' in a post-state. For example, `States[P]` = **tuple of** `val:Int, toSend:Set[Int]`, and `P.val` is a term of sort `Int` equivalent to the state variable `val`.

If an automaton A has type parameters, the notation for its aggregate state sort is more complicated, because there can be different instantiations of A with different actual types, and a simple notation $States[A]$ for the aggregate state sort would be ambiguous. To avoid this ambiguity, IOA includes the type parameters of A (if any) in the notation $States[A, types^A]$ for the aggregate state sort of A, and the aggregate state and post-state variables A have this sort $States[A, types^A]$. For example, `States[Channel,Node,Msg]` = **tuple of** `contents:Set[Msg]`, and `Channel.contents` is a term of sort `Set[Msg]` equivalent to the state variable `contents`.

As we will see in Section 13.2, including type parameters in the name of the aggregate state sort enables us to generate distinct aggregate state sorts for each instantiation of A.

11.2.2 Local variables

In previous editions of the language, IOA introduced hidden action parameters with the keyword **choose** appearing subsequent to the **where** clause. Thus, hidden or **choose** parameters could not appear in the **where** clause. In the course of writing this document, we discovered a need for hidden parameters in the **where** clauses of desugared input actions (see Chapter 12). In addition, we believed that the ability to assign (temporary) values to hidden parameters would simplify the definitions of expanded transition definitions of composite automata.[7] We introduced local variables into IOA to serve both these purposes. Local variables replace and extend **choose** parameters.

[6] An unambiguous variable identifier can be used alone. If two variables defined in the same scope have the same identifier, but different sorts, their identifier may need to be qualified by their sorts.

[7] In the end, our final definitions in Sections 15.6–15.9 do not to use this feature. However, the ability to assign to local variables was deemed useful and remains in the language.

Thus, the keyword **local** replaces the keyword **choose** in transition definition parameter lists and local variables are those introduced following the keyword **local** in these parameter lists.

In the new notation, the scope of local variables extends to the whole transition definition, not just to the precondition and effects. In addition, local variables may be assigned values in the **eff** clause. Semantically, local variables are *not* part of the state of the I/O automaton represented by an IOA program. Rather, they define *intermediate states* that occur during the execution of an atomic transitions, but are not visible externally. Therefore, local variables may not appear in simulation relations or invariants.

Although local variables differ significantly from state variables in terms of semantics, their syntactic treatment is similar. As for state variables, IOA automatically defines an *aggregate local sort*, together with *aggregate local* and *post-local* variables, to provide a second, equivalent notation for references to local and post-local variables. For every transition definition t_j for an action π of type *kind* in automaton A, the aggregate local sort $Locals[A, types^A, kind, \pi, t_j]$ is a tuple sort with a selection operator $\_\_.v{:}States[A] \rightarrow T$ for each local variable v of sort T. Furthermore, aggregate local and post-local variables, A and A' of sort $localVars^{A,\pi}_{kind,t_j}$, are defined in the scope of that transition definition. If there is only one transition definition for an action π of type *kind*, we omit t_j in the notation for this sort. For example, the aggregate locals sort $Locals$[`P,out,overflow`] is **tuple of** `t:Set[Int]`, and `P.t` is a term of sort `Set[Int]` equivalent to the local variable `t` in the scope of `overflow`.

Note that the automaton name A is used as the identifier for *two* aggregate variables in every transition definition: $A{:}States[A, types^A]$ and $A{:}Locals[A, types^A, kind, \pi, t_j]$. As specified in Section 11.3, $stateVars^A$ and $localVars^{A,\pi}_{kind,t_j}$ must have no variables in common. Therefore, the aggregate sorts have no selection operators in common and there is no ambiguity.

The initial values of local variables are constrained by the **where** predicate of the declaring transition definition. In particular, a transition *kind* $\pi(\dots)$ **case** t_j is defined only for values of its parameters that

1. satisfy the **where** clause of that *kind* of π in the signature of A, and

2. together with some choice of initial values for its local variables, satisfy the **where** clause of the transition definition.

A transition is *enabled* only for the values of its parameters and local variables for which it is defined and for which the precondition, if any, is satisfied.

Thus, the initial values of local variables are chosen nondeterministically from among the values that meet these constraints. Local variables serve as hidden parameters with the semantics formerly applied to **choose** parameters. We provide a formal treatment of the "values of its parameters" and "some choice of values" at the end of Section 12.

LOCATION OF TERM	VARIABLES THAT CAN OCCUR FREELY IN TERM
$params^A$	$vars^A$
$params^{A,\pi}_{kind}$	$vars^A$, $vars^{A,\pi}_{kind}$
$P^{A,\pi}_{kind}$	$vars^A$, $vars^{A,\pi}_{kind}$
$initVals^A$	$vars^A$
P^A_{init}	$vars^A$, $stateVars^A$
$params^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$
$P^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$
$Pre^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, $stateVars^A$
$Prog^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, $stateVars^A$
$ensuring^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, $stateVars^A$, $postVars^A$, $localPostVars^{A,\pi}_{kind,t_j}$

Table 11.1: Variables that can occur freely in terms in the definition of a primitive automaton. Variables listed on the right may occur freely in the syntactic structure listed to their left.

Example 11.1 The **type** declarations and variables automatically defined for the sample automata `Channel`, `P`, and `Watch` are shown in Figure 11.2.

```
type States[Channel,Node,Msg] = tuple of contents:Set[Msg]
type States[P] = tuple of val:Int, toSend:Set[Int]
type States[Watch,T] = tuple of seen:Array[T,Bool]
type Locals[P,out,overflow] = tuple of t:Set[Int]

Channel: States[Channel, Node, Msg]
P: States[P]
Watch: States[Watch,T]
P: Locals[P,out,overflow]
```

Figure 11.2: Automatically defined types and variables for sample automata

11.3 Static semantic checks

The following conditions must be true for an IOA program to represent a valid primitive I/O automaton. These conditions, which can be checked statically, are currently performed by `ioaCheck`, the IOA parser and static-semantic checker.

- ✓ No sort appears more than once in $types^A$.
- ✓ Each action name (*e.g.*, π) occurs at most three times in the signature of an automaton: at most once in a list of input actions, at most once in a list of output actions, and at most once in a list of internal actions.

- ✓ Each occurrence of an action name (*e.g.*, π) in the signature of an automaton, or in one of its transition definitions, must be followed by the same number and sorts of parameters.

- ✓ The sequences $vars^A$ and $vars^{A,\pi}_{kind}$ of variables contain no duplicates; furthermore, no variable appears in both $vars^A$ and $vars^{A,\pi}_{kind}$ for any value of *kind*.[8]

- ✓ For each transition definition t_j for an action of type *kind* named π, no variable appears more than once in the combination of the sequences $vars^A$, $stateVars^A$, $postVars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, and $localPostVars^{A,\pi}_{kind,t_j}$.

- ✓ For each transition definition t_j for an action of type *kind* named π, and for any identifier v and sort S, the sequences $stateVars^A$ and $localVars^{A,\pi}_{kind,t_j}$ do not contain both of the variables $v{:}S$ and $v'{:}S$.

- ✓ Any operator that occurs in a term used in the definition of an automaton must be introduced by a type definition or **axioms** clause in the IOA specification that contains the automaton definition, by a theory specified in the **assumes** clause of the definition, or by a built-in datatype of IOA.

- ✓ Any variable that occurs freely in a term used in the definition of an automaton must satisfy the restrictions imposed by Table 11.1.

11.4 Semantic proof obligations

The following conditions must also be true for an IOA program to represent a valid I/O automaton. Except in special cases, these conditions cannot be checked automatically, because they may require nontrivial proofs (or even be undecidable); hence static semantic checkers must translate all but the simplest of them into proof obligations for an automated proof assistant. These proof obligations must be discharged using the axioms provided by IOA's built-in types, by the theories associated with the type definitions and the **axioms** in the IOA specification that contains the automaton definition, and by the theories associated with the **assumes** clause of that definition.

- ✓ The sets of input, output, and internal actions in an I/O automaton must be disjoint. Thus, for each sequence of values for the parameters of an action named π in the definition of an automaton A, at most one of $P^{A,\pi}_{in}$, $P^{A,\pi}_{out}$, and $P^{A,\pi}_{int}$ can be true.

[8] This restriction is designed to avoid the confusion that would result if variables in $vars^{A,\pi}_{kind}$ are allowed to hide or override variables with the same identifiers and sorts in $vars^A$. A stronger restriction would prohibit an identifier from appearing in two different variables (of different sorts) in $vars^A$ and $vars^{A,\pi}_{kind}$; this restriction would avoid the need to pick a fresh variable when an instantiation of A causes two variables with the same identifier to clash by mapping their sorts to a common sort. However, IOA does not make this stronger restriction.

Special cases arise if two of the three signature **where** clauses for π are literally *false* or if two of three clauses are literally *true*. In the former case, the check automatically succeeds; in the latter, it automatically fails.

✓ There must be a transition defined for every action specified in the signature. Thus, for each sequence of values for the parameters of an action named π that make $P^{A,\pi}_{kind}$ true, there must be a transition definition t_j for π of type *kind* such that $P^{A,\pi}_{kind,t_j}$ is true for these values together with some values for the local variable of that transition definition.

✓ For each *kind* of each action π, at most one transition definition t_j can be defined for each sequence of parameters values. That is, for each sequence of values, $P^{A,\pi}_{kind,t_j}$ can be true for at most one value of j.

Special cases arise if all but one of the transition definition **where** clauses for a kind of an action are literally *false* or any two are literally *true*. In the former case, the check automatically succeeds; in the latter, it automatically fails.

We define these proof obligations more formally at the end of Chapter 12.

Chapter 12

Desugaring primitive automata

A spoonful of sugar helps the medicine go down.

— Mary Poppins [109]

The syntax for IOA programs described in Chapter 11 allows some flexibility of expression. However, when defining semantic checks and algorithmic manipulations (*e.g.*, composition) of IOA programs, it is helpful to restrict attention, without loss of generality, to IOA programs that conform to a more limited syntax.

In this chapter, we describe how to transform any primitive IOA program (as in Figure 11.1) into an equivalent program (Figure 12.7) written with a more limited syntax. We describe this transformation in four stages. First, in Section 12.1, we show how to *desugar* terms that appear as parameters by replacing them with variables constrained by **where** clauses; that is, we show how to reformulate action and transition definitions so as to eliminate the use of terms as parameters. Second, in Section 12.2, we show how to introduce *canonical parameters* into desugared actions and transition definitions. A canonicalized action is parameterized by the same sequence of variables in all appearances, both in the signature and in the transition definitions. Third, in Section 12.3, we *combine* all transition definitions of a single kind of an action into a single transition definition. Fourth, in Section 12.4, we convert each reference to a state variable x to the equivalent reference $A.x$ defined in Section 11.2. In Section 12.5, we summarize the effects of these desugarings, which are illustrated in Figure 12.7. Finally, in Section 12.6, we use the result of the first two transformations to formalize the semantic proof obligations introduced in Chapter 11.

automaton $A(types^A, vars^A)$

signature

...

input $\pi(vars^{A,\pi}_{in,desug})$ **where** $P^{A,\pi}_{in} \ \wedge \ vars^{A,\pi}_{in,desug} = params^{A,\pi}_{in}$

output $\pi(vars^{A,\pi}_{out,desug})$ **where** $P^{A,\pi}_{out} \ \wedge \ vars^{A,\pi}_{out,desug} = params^{A,\pi}_{out}$

internal $\pi(vars^{A,\pi}_{int,desug})$ **where** $P^{A,\pi}_{int} \ \wedge \ vars^{A,\pi}_{int,desug} = params^{A,\pi}_{int}$

...

states $stateVars^A := initVals^A$ **initially** P^A_{init}

transitions

...

input $\pi(vars^{A,\pi}_{in,t_j,desug};$ **local** $localVars^{A,\pi}_{in,t_j}, vars^{A,\pi}_{in,t_j})$ **case** t_j

where $P^{A,\pi}_{in,t_j} \ \wedge \ vars^{A,\pi}_{in,t_j,desug} = params^{A,\pi}_{in,t_j}$

eff $Prog^{A,\pi}_{in,t_j}$ **ensuring** $ensuring^{A,\pi}_{in,t_j}$

output $\pi(vars^{A,\pi}_{out,t_j,desug};$ **local** $localVars^{A,\pi}_{out,t_j}, vars^{A,\pi}_{out,t_j})$ **case** t_j

where $P^{A,\pi}_{out,t_j} \ \wedge \ vars^{A,\pi}_{out,t_j,desug} = params^{A,\pi}_{out,t_j}$

pre $Pre^{A,\pi}_{out,t_j}$

eff $Prog^{A,\pi}_{out,t_j}$ **ensuring** $ensuring^{A,\pi}_{out,t_j}$

internal $\pi(vars^{A,\pi}_{int,t_j,desug};$ **local** $localVars^{A,\pi}_{int,t_j}, vars^{A,\pi}_{int,t_j})$ **case** t_j

where $P^{A,\pi}_{int,t_j} \ \wedge \ vars^{A,\pi}_{int,t_j,desug} = params^{A,\pi}_{int,t_j}$

pre $Pre^{A,\pi}_{int,t_j}$

eff $Prog^{A,\pi}_{int,t_j}$ **ensuring** $ensuring^{A,\pi}_{int,t_j}$

...

Figure 12.1: Preliminary form of a desugared primitive automaton: all action parameters are variables

12.1 Desugaring terms used as parameters

12.1.1 Signature

We desugar **const** parameters for an action in A's signature by introducing fresh variables and modifying the action's **where** clause. For each **const** parameter we introduce a fresh variable and add a conjunct to the **where** clause that equates the new variable with the term that served as the **const** parameter. For example, if t is a term of sort T, then we desugar the action

input $\pi(vars^{A,\pi}_{in},$ **const** $t)$ **where** $P^{A,\pi}_{in}$

as

input $\pi(vars^{A,\pi}_{in}, v{:}T)$ **where** $v = t \ \wedge \ P^{A,\pi}_{in}$

Here, $v{:}T$ is a fresh variable, that is, one that does not appear in $vars^A$, $vars^{A,\pi}_{in}$, $stateVars^A$, $postVars^A$, $localVars^{A,\pi}_{in,t_j}$, or $localPostVars^{A,\pi}_{in,t_j}$ for any j.[1]

Let $P^{A,\pi}_{kind,desug}$ be the **where** predicate that results after all **const** parameters in $params^{A,\pi}_{kind}$ have been desugared. Let $vars^{A,\pi}_{kind,desug}$ be the sequence of distinct variables that parameterize π after desugaring. Note that all variables that occur freely in $P^{A,\pi}_{kind,desug}$ are either in $vars^{A,\pi}_{kind,desug}$ or in $vars^A$. In general, $vars^{A,\pi}_{kind,desug}$ is a supersequence of $vars^{A,\pi}_{kind}$ (in that it contains a fresh variable for each **const** parameter in $params^{A,\pi}_{kind}$). In the above example, a **const** parameter appears in the last position of $params^{A,\pi}_{in}$. In general, **const** parameters may appear in any position. A fresh variable appears in $vars^{A,\pi}_{kind,desug}$ in the same position the **const** parameter it replaces appears in $params^{A,\pi}_{kind}$.

The preliminary form for desugaring an automaton signature shown in Figure 12.1 indicates that each variable in $vars^{A,\pi}_{kind,desug}$ is equated to the corresponding entry in $params^{A,\pi}_{kind}$. (In the figure, we use $params^{A,\pi}_{kind}$ to mean the sequence of terms without the **const** keyword.) An obvious simplification is to omit any identity conjuncts that arise when a variable in $vars^{A,\pi}_{kind}$ is equated to itself.

12.1.2 Transition definitions

We desugar the parameters for each transition definition for an action named π to eliminate parameters that are not just simple variable references.[2] As shown in Figure 12.1, we first replace the transition parameters $params^{A,\pi}_{kind,t_j}$ by references to distinct fresh variables $vars^{A,\pi}_{kind,t_j,desug}$, that is, to variables that do not appear in $vars^A$, $stateVars^A$, $postVars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, or $localPostVars^{A,\pi}_{kind,t_j}$.[3] Second, we maintain the original semantics of the transition definition by adding conjuncts to the **where** clause to equate the new variables with the old parameters. Third, because transition definition parameters may introduce variables implicitly, but **where** clauses may not, we introduce the previously free variables (*i.e.*, $vars^{A,\pi}_{kind,t_j}$) as additional local variables, letting $localVars^{A,\pi}_{kind,t_j,desug}$ be the concatenation of $localVars^{A,\pi}_{kind,t_j}$ and $vars^{A,\pi}_{kind,t_j}$. In effect, these steps move terms used as parameters into the **where** clause. For example, if t is a term and v is a fresh

[1] For the purposes of this transformation, it suffices to pick some $v{:}T$ that does not appear in either $vars^A$ or $vars^{A,\pi}_{in}$. However, by ensuring that $v{:}T$ is distinct from additional variables, we avoid having to replace it by yet another fresh variable when we introduce canonical transition parameters, as described in Section 12.2. Furthermore, to avoid any ambiguity that may arise when two variables share an identifier, and to avoid having to replace $v{:}T$ by yet another fresh variable in an instantiation of A that maps T and the sort of another variable with identifier v to a common sort, it is helpful to pick v to be an identifier that does not appear in $vars^A$, $vars^{A,\pi}_{in}$, $stateVars^A$, $postVars^A$, $localVars^{A,\pi}_{in,t_j}$, or $localPostVars^{A,\pi}_{in,t_j}$ for any j.

[2] As mentioned in Footnote 1, we distinguish between action parameters in the signature that are terms (**const** parameters) and those that are variable declarations to provide strong typing for variable declarations. Since the sorts of $params^{A,\pi}_{kind}$ determine the sorts of $params^{A,\pi}_{kind,t_j}$, there is no need for such a distinction in transition parameters.

[3] It suffices to replace just those parameters that are not simply references to variables, because the fresh variables corresponding to such terms disappear when we substitute references to canonical variables for the parameters, as described in the next section. However, the replacement is easier to describe if we replace all parameters.

Furthermore, as for **const** parameters, to avoid any ambiguity that may arise in the **where** clause when two variables share an identifier, and to avoid having to replace $v{:}T$ by yet another fresh variable in an instantiation of A that maps T and the sort of another variable with identifier v to a common sort, it is helpful to pick v to be an identifier that is not in $vars^A$, $stateVars^A$, $postVars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, or $localPostVars^{A,\pi}_{kind,t_j}$.

variable with the same sort as t, then we desugar the transition definition

input $\pi(t)$ **where** $P^{A,\pi}_{in,t_j}$

as

input $\pi(v; \textbf{local } vars^{A,\pi}_{in,t_j})$ **where** $v = t \ \wedge \ P^{A,\pi}_{in,t_j}$

Let $P^{A,\pi}_{kind,t_j,desug}$ be the **where** predicate that results after transition parameters have been desugared in this fashion. Then any variable that has a free occurrence in this predicate must be in $vars^A$, $vars^{A,\pi}_{kind,t_j,desug}$, or $localVars^{A,\pi}_{kind,t_j,desug}$.

After **const** and transition definition terms have been desugared, the valid occurrences of free variables in syntactic forms, shown in Table 11.1, is revised by those shown in Table 12.1. After desugaring, $params^{A,\pi}_{kind} = vars^{A,\pi}_{kind,desug}$ and $params^{A,\pi}_{kind,t_j} = vars^{A,\pi}_{kind,t_j,desug}$.

LOCATION OF TERM	VARIABLES THAT CAN OCCUR FREELY IN TERM
$P^{A,\pi}_{kind,desug}$	$vars^A$, $vars^{A,\pi}_{kind,desug}$
$P^{A,\pi}_{kind,t_j,desug}$	$vars^A$, $vars^{A,\pi}_{kind,t_j,desug}$, $localVars^{A,\pi}_{kind,t_j,desug}$

Table 12.1: Variables that can occur freely in terms in the definition of a desugared primitive automaton. Variables listed on the right may occur freely in the syntactic structure listed to their left.

Example 12.1 The first step in desugaring the primitive automata defined in Figures 10.1–10.3 is shown in Figure 12.2. For the automaton `Channel`, `n1:Node` and `n2:Node` are fresh variables introduced to desugar the **const** parameters in the signature. Similarly, `n1:Node`, `n2:Node`, and `m1:Msg` are fresh variables introduced to desugar transition parameters. Since both $vars^{\texttt{Channel},\texttt{send}}_{in,t_1}$ and $vars^{\texttt{Channel},\texttt{receive}}_{out,t_1}$ contain the single variable `m:Msg`, we introduce `m:Msg` as a local variable for each transition definition. Notice that the variables introduced for each action need be fresh only with respect to `i:Node`, `j:Node`, and `m:Msg`; furthermore, "freshness" need not extend across transitions or between actions and transitions.

The automata `P` and `Watch` are desugared in a similar fashion. Since there are no **const** parameters in the signature of `Watch`, that signature is unchanged. Since the parameters for the transition definitions for the `overflow` action in `Watch` contain two free variables, `x` and `s`, the desugared transition definitions declare these variables as local. Also, in the second of the desugared transition definitions, the desugared **where** clause incorporates the original **where** clause as a conjunct.

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
    output receive(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
  states contents:Set[Msg] := {}
  transitions
    input send(n1, n2, m1; local m:Msg) where n1 = i ∧ n2 = j ∧ m1 = m
      eff contents := insert(m, contents)
    output receive(n1, n2, m1; local m:Msg)
             where n1 = i ∧ n2 = j ∧ m1 = m
      pre m ∈ contents
      eff contents := delete(m, contents)

automaton P(n:Int)
  signature
    input receive(i1, i2, x:Int) where i1 = n-1 ∧ i2 = n
    output send(i1, i2, x:Int) where i1 = n ∧ i2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(i1, i2, i3; local x:Int)
      where i1 = n-1 ∧ i2 = n ∧ i3 = x
      eff ...    % effect clause unchanged from original definition of P
    output send(i1, i2, i3; local x:Int)
      where i1 = n ∧ i2 = n+1 ∧ i3 = x
      pre x ∈ toSend
      eff toSend := delete(x, toSend)
    output overflow(i1, s1; local t, s:Set[Int]) where i1 = n ∧ s1 = s
      pre s = toSend ∧ n < size(s) ∧ t ⊆ s
      eff toSend := t

automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(t1, s1; local x:T, s:Set[T])
             where t1 = x ∧ s1 = s ∪ {x}
      eff seen[x] := true
    input overflow(t1, s1; local x:T, s:Set[T])
             where ¬(x ∈ s) ∧ t1 = x ∧ s1 = s
      eff seen[x] := false
    output found(t1; local x:T) where t1 = x
      pre seen[x]
```

Figure 12.2: Preliminary desugarings of the sample automata `Channel`, `P`, and `Watch`

12.2 Introducing canonical names for parameters

12.2.1 Signature

IOA does not require that the sequences of variables $vars_{in}^{A,\pi}$, $vars_{out}^{A,\pi}$, and $vars_{int}^{A,\pi}$ be the same. For example, **const** parameters may cause these sequences to have different lengths. However, since IOA requires $params_{in}^{A,\pi}$, $params_{out}^{A,\pi}$, and $params_{int}^{A,\pi}$ to contain the same number and sorts of elements, the desugared versions of these sequences (*i.e.*, $vars_{in,desug}^{A,\pi}$, $vars_{out,desug}^{A,\pi}$, and $vars_{int,desug}^{A,\pi}$) do have the same number and sorts of elements. We choose one of these desugared variable sequences to be the *canonical parameters* for the action π in A. We call the canonical sequence $vars^{A,\pi}$. We replace the other two sequences of parameters for π in the signature of A by $vars^{A,\pi}$, and we define substitutions $\sigma_{kind}^{A,\pi}$ to replace $vars_{kind,desug}^{A,\pi}$ with $vars^{A,\pi}$ in $P_{kind}^{A,\pi}$.[4]

12.2.2 Transition definitions

We canonicalize the parameters for each transition definition for an action named π so that the definition also uses $vars^{A,\pi}$ as its parameters. Specifically, we replace the references to variables that parameterize a desugared transition definition of π (*i.e.*, $vars_{kind,t_j,desug}^{A,\pi}$) by references to the canonical variables (*i.e.*, $vars^{A,\pi}$) throughout the transition definition. Therefore we define a substitution $\sigma_{kind,t_j}^{A,\pi}$ to perform this replacement and apply it to the whole transition definition. As described in Chapter 17, if the canonical variables clash with the desugared local variables (*i.e.*, $localVars_{kind,t_j,desug}^{A,\pi}$), we must substitute fresh local variables for those that clash. The variables introduced by the substitution must be be distinct and fresh with respect to $vars^{A}$, $vars^{A,\pi}$, and the desugared local variables. The substitutions for canonicalization are listed in Table 12.2. Variables listed in the center column are mapped by the substitution named in the left column to those listed in the right column.

12.2.3 Simplifying local variables

Finally, we simplify each desugared and canonicalized transition definition for actions named π by eliminating extraneous local variables. A local variable may be eliminated if it is never an lvalue in an assignment in the transition definition for π and if the **where** clause equates it with a canonical variable for π, that is, if it is used only as a constant in the transition definition and is already named by a canonical parameter.

This simplification is accomplished in four steps.

[4] See Chapter 17 for a precise definition of a substitution, which maps a set of variables to a set of terms. Often we represent the domain and range of a substitution as sequences, with the ith variable in the domain being replaced by the ith variable or term in the range.

automaton $A(types^A, vars^A)$

signature

...

input $\pi(vars^{A,\pi})$ **where** $\sigma_{in}^{A,\pi}(P_{in,desug}^{A,\pi})$

output $\pi(vars^{A,\pi})$ **where** $\sigma_{out}^{A,\pi}(P_{out,desug}^{A,\pi})$

internal $\pi(vars^{A,\pi})$ **where** $\sigma_{int}^{A,\pi}(P_{int,desug}^{A,\pi})$

...

states $stateVars^A := initVals^A$ **initially** P_{init}^A

transitions

$$\sigma_{in,t_j}^{A,\pi}\left[\begin{array}{l}\textbf{input } \pi(vars_{in,t_j,desug}^{A,\pi};\ \textbf{local } localVars_{in,t_j,desug}^{A,\pi}) \textbf{ case } t_j \textbf{ where } P_{in,t_j,desug}^{A,\pi} \\ \quad \textbf{eff } Prog_{in,t_j}^{A,\pi} \textbf{ ensuring } ensuring_{in,t_j}^{A,\pi}\end{array}\right]$$

$$\sigma_{out,t_j}^{A,\pi}\left[\begin{array}{l}\textbf{output } \pi(vars_{out,t_j,desug}^{A,\pi};\ \textbf{local } localVars_{out,t_j,desug}^{A,\pi}) \textbf{ case } t_j \textbf{ where } P_{out,t_j,desug}^{A,\pi} \\ \quad \textbf{pre } Pre_{out,t_j}^{A,\pi} \\ \quad \textbf{eff } Prog_{out,t_j}^{A,\pi} \textbf{ ensuring } ensuring_{out,t_j}^{A,\pi}\end{array}\right]$$

$$\sigma_{int,t_j}^{A,\pi}\left[\begin{array}{l}\textbf{internal } \pi(vars_{int,t_j,desug}^{A,\pi};\ \textbf{local } localVars_{int,t_j,desug}^{A,\pi}) \textbf{ case } t_j \textbf{ where } P_{int,t_j,desug}^{A,\pi} \\ \quad \textbf{pre } Pre_{int,t_j}^{A,\pi} \\ \quad \textbf{eff } Prog_{int,t_j}^{A,\pi} \textbf{ ensuring } ensuring_{int,t_j}^{A,\pi}\end{array}\right]$$

...

Figure 12.3: Intermediate form of a desugared primitive automaton with canonical action parameters (cf. Figure 12.1)

SUBSTITUTION	DOMAIN	RANGE
$\sigma_{kind}^{A,\pi}$	$vars_{kind,desug}^{A,\pi}$	$vars^{A,\pi}$
$\sigma_{kind,t_j}^{A,\pi}$	$vars_{kind,t_j,desug}^{A,\pi}$	$vars^{A,\pi}$
$\sigma_{kind,t_j,simp}^{A,\pi}$	Redundant variables in $\sigma_{kind,t_j}^{A,\pi}(localVars_{kind,t_j,desug}^{A,\pi})$	$vars^{A,\pi}$
σ^A	$x \in stateVars^A$	$A{:}States[A, types^A].x$
	$x' \in postVars^A$	$A'{:}States[A, types^A].x$
	$x \in localVars_{kind,t_j}^{A,\pi}$	$A{:}Locals[A, types^A, \pi].x$
	$x' \in localPostVars_{kind,t_j}^{A,\pi}$	$A'{:}Locals[A, types^A, \pi].x$

Table 12.2: Substitutions used in desugaring a primitive automaton. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

1. Define a substitution $\sigma^{A,\pi}_{kind,t_j,simp}$ that maps the redundant local variables to the corresponding canonical variables.

2. Apply $\sigma^{A,\pi}_{kind,t_j,simp}$ to each clause in the transition definition: the **where**, **pre**, **eff**, and **ensuring** clauses.

3. Delete identity conjuncts from the **where** clause.

4. Delete the declarations of local variables that no longer appear in the transition.

Example 12.2 The second step in desugaring the primitive automata defined in Figures 10.1–10.3 is shown in Figure 12.4. The definitions in this figure are obtained from those in Figure 12.2 by selecting canonical parameters for each action.

Since each action occurs only once in the signature of the automaton `Channel`, selecting the canonical variables is trivial:

- $vars^{\texttt{Channel},\texttt{send}}$ defaults to $vars^{\texttt{Channel},\texttt{send}}_{in,desug} = \langle$`n1:Node, n2:Node, m:Msg`$\rangle$, and
- $vars^{\texttt{Channel},\texttt{receive}}$ defaults to $vars^{\texttt{Channel},\texttt{receive}}_{out,desug} = \langle$`n1:Node, n2:Node, m:Msg`$\rangle$.

These selections also make canonicalizing the signature trivial, because identity substitutions suffice.

We canonicalize the transition definitions by defining two substitutions.

- $\sigma^{\texttt{Channel},\texttt{send}}_{in,t_1}$ maps $vars^{\texttt{Channel},\texttt{send}}_{in,t_1,desug} = \langle$`n1:Node`, `n2:Node`, `m1:Msg`$\rangle$, to $vars^{\texttt{Channel},\texttt{send}}$ by replacing the parameter `m1:Msg` with the canonical parameter `m:Msg`. To avoid a conflict between the local variable `m:Msg` and the canonical parameter `m:Msg`, the substitution also replaces `m:Msg` by the fresh variable `m2:Msg`.
- In the same way, $\sigma^{\texttt{Channel},\texttt{receive}}_{out,t_1}$ maps $vars^{\texttt{Channel},\texttt{receive}}_{out,t_1,desug} = \langle$`n1:Node`, `n2:Node`, `m1:Msg`$\rangle$ to $vars^{\texttt{Channel},\texttt{receive}}$ by replacing the parameter `m1:Msg` with the canonical parameter `m:Msg` and the local variable `m:Msg` with the fresh variable `m2:Msg`.

Applying these substitution to the transition definitions produces

```
input send(n1, n2, m; local m2:Msg) where n1 = i ∧ n2 = j ∧ m = m2
  eff contents := insert(m2, contents)
output receive(n1, n2, m; local m2:Msg) where n1 = i ∧ n2 = j ∧ m = m2
  pre m2 ∈ contents
  eff contents := delete(m2, contents)
```

However, the local variable `m2` is extraneous in both transition definitions, because it is equated with `m` in the **where** clause and no value is assigned to it. Hence `m2` equals `m` throughout the transition, and we can eliminate it entirely by applying a substitution (*e.g.*, $\sigma^{\texttt{channel},\texttt{send}}_{in,t_1,simp}$, which maps `m2` to `m`) to the **where**,**eff** and **pre** (in the case of `receive`) clauses and simplifying the result, as shown in Figure 12.4.

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
    output receive(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
  states contents:Set[Msg] := {}
  transitions
    input send(n1, n2, m) where n1 = i ∧ n2 = j
      eff contents := insert(m, contents)
    output receive(n1, n2, m) where n1 = i ∧ n2 = j
      pre m ∈ contents
      eff contents := delete(m, contents)

automaton P(n:Int)
  signature
    input receive(i1, i2, x:Int) where i1 = n-1 ∧ i2 = n
    output send(i1, i2, x:Int) where i1 = n ∧ i2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(i1, i2, x) where i1 = n-1 ∧ i2 = n
      eff if val = 0 then val := x
          elseif x < val then
            toSend := insert(val, toSend);
            val := x
          elseif val < x then
            toSend := insert(x, toSend)
          fi
    output send(i1, i2, x) where i1 = n ∧ i2 = n+1
      pre x ∈ toSend
      eff toSend := delete(x, toSend)
    output overflow(i1, s; local t:Set[Int]) where i1 = n
      pre s = toSend ∧ n < size(s) ∧ t ⊆ s
      eff toSend := t

automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[T]) where s = s2 ∪ {x}
      eff seen[x] := true
    input overflow(x, s) where ¬(x ∈ s)
      eff seen[x] := false
    output found(x)
      pre seen[x]
```

Figure 12.4: Intermediate desugarings of the sample automata `Channel`, `P`, and `Watch`, obtained from the preliminary desugarings in Figure 12.2 by selecting canonical parameters for each action

As for `Channel`, each action occurs only once in the signature of the automaton `P`. Hence, it is trivial to select $vars^{\texttt{P,receive}}$, $vars^{\texttt{P,send}}$, and $vars^{\texttt{P,overflow}}$ and to canonicalize the signature.

To map $vars^{\texttt{P,receive}}_{in,t_1,desug}$ (*i.e.*, ⟨`i1:Int`, `i2:Int`, `i3:Int`⟩) to $vars^{\texttt{P,receive}}$, we define $\sigma^{\texttt{P,receive}}_{in,t_1}$ to replace `i3:Int` by `x:Int`. To avoid conflicts between the local variable `x:Int` and the canonical parameter `x:Int`, the substitution also replaces `x:Int` by `i4:Int`. Applying this substitution to the transition definition produces:

```
input receive(i1, i2, x; local i4:Int) where i1 = n-1 ∧ i2 = n ∧ x = i4
  eff if val = 0 then val := i4
      elseif i4 < val then
        toSend := insert(val, toSend);
        val := i4
      elseif val < i4 then
        toSend := insert(i4, toSend)
      fi
```

Since the local variable `i4` equals `x` throughout the transition definition, we can eliminate it entirely by defining a substitution mapping `i4` to `x`, applying that substitution to the **where** and **eff** clauses, and simplifying the result, as shown in Figure 12.4.

Canonicalization of the `send` transition follows the same pattern as the `receive` transition. Application of the canonicalizing substitution $\sigma^{\texttt{P,send}}_{out,t_1}$ yields:

```
output send(i1, i2, x; local i4:Int) where i1 = n ∧ i2 = n+1 ∧ x = i4
pre i4 ∈ toSend
eff toSend := delete(i4, toSend)
```

This definition simplifies to the one shown in Figure 12.2, which does not contain a local variable.

Similarly applying the canonicalizing substitution $\sigma^{\texttt{P,overflow}}_{out,t_1}$ to the `overflow` transition yields:

```
output overflow(i1, s; local t, s2:Set[Int]) where i1 = n ∧ s = s2
  pre s2 = toSend ∧ n < size(s2) ∧ t ⊆ s2
  eff toSend := t
```

Once again, this definition simplifies to the one shown in Figure 12.2. Notice that the local variable `t` cannot be eliminated because it is not equated with a canonical parameter. Further notice that, in this case, canonicalization has eliminated all the local variables introduced in the desugaring step.

As for `Channel` and `P`, each action occurs only once in the signature of the automaton `Watch`. Hence it is trivial to select $vars^{\texttt{Watch,overflow}}$ and $vars^{\texttt{watch,found}}$.

Canonicalizing the two transition definitions for `overflow` proceeds by defining $\sigma^{\texttt{watch,overflow}}_{in,t_1}$ and $\sigma^{\texttt{watch,overflow}}_{in,t_2}$, which happen to be the same. They map `t1:T` to `x:T`, `s1:Set[T]` to `s:Set[T]`, `s:Set[T]` to `s2:Set[T]`, and `x:T` to `t2:T`. Applying these substitutions to the transition definitions yields:

```
input overflow(x, s; local t2:T, s2:Set[T])
          where x = t2 ∧ s = s2 ∪ {t2}
  eff seen[t2] := true
```

```
input overflow(x, s; local t2:T, s2:Set[T])
          where ¬(t2 ∈ s2) ∧ x = t2 ∧ s = s2
  eff seen[x] := false
```

The local variable `t2:T` can be eliminated from both transition definitions. The local variable `s2:Set[T]` can be eliminated from the second transition definition but *not* from the first. These simplifications result in the transition definitions shown in Figure 12.4.

Notice that after the simplification of the local variable, the semantic meaning of the parameter `s:Set[T]` in the desugared and canonicalized automaton shown in Figure 12.4 is different than the meaning of the parameter `s:Set[T]` in the original automaton shown in Figure 10.3. The parameter `s:Set[T]` in the original actually corresponds to the local variable `s2:Set[T]` in the canonicalized version.

Applying the canonicalizing substitution $\sigma_{in,t_1}^{\texttt{watch,found}}$ to the `found` transition yields:

```
output found(x; local t2:T) where x = t2
  pre seen[t2]
```

After its local variables are simplified, the transition definition shown in Figure 12.4 is identical to the one originally defined in Figure 10.3.

12.3 Combining transition definitions

We will see in Sections 15.7–15.9 that combining multiple transition definitions for a given action into a single transition definition is useful for composing automata. It is necessary for combining input actions that execute atomically in the composition, and it avoids a code explosion multiplicative in the number of input and output actions. Because this transition combining step is easy to understand when applied to a single primitive automaton, we describe it here and assume all automata hereafter have only a single transition definition per kind per action, as shown in Figure 12.5. To combine the transition definitions for a given *kind* of an action π, we need to combine their sequences of parameters, their local variables, and their **where**, **pre**, **eff**, and **ensuring** clauses into one, semantically equivalent, transition definition.

Furthermore, as will be discussed further in Chapter 15, the kind of an action may be changed by composition. Input actions may be subsumed by output actions, and output actions may be hidden as internal actions. Thus, the expansion of a composite automaton may combine transition definitions across kinds. To facilitate such combinations, we collect together all the local variables for each action of an automaton A into a single sequence of variables $localVars^{A,\pi}$, which is the concatenation (with duplicates removed) of the all sequences $localVars_{kind,t_j}^{A,\pi}$. Again, this variable combining step is easy to understand when applied to a single primitive automaton, so we describe it here and assume all automata hereafter have only one sequence of local variables per action name.

In describing this combination, we assume that parameters of the automaton have already been

automaton $A(types^A, vars^A)$

...

states $stateVars^A := initVals^A$ **initially** $\sigma^A(P^A_{init})$

transitions

input $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi})$ **where** $\bigvee_j P^{A,\pi}_{in,t_j,desug}$

eff

if $P^{A,\pi}_{in,t_j,desug}$ **then** $Prog^{A,\pi}_{in,t_j,desug}$

elseif ...

fi

ensuring $\bigwedge_j \left(P^{A,\pi}_{in,t_j,desug} \Rightarrow ensuring^{A,\pi}_{in,t_j,desug}\right)$

output $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi})$ **where** $\bigvee_j P^{A,\pi}_{out,t_j,desug}$

pre $\bigvee_j \left(P^{A,\pi}_{out,t_j,desug} \wedge Pre^{A,\pi}_{out,t_j,desug}\right)$

eff

if $P^{A,\pi}_{out,t_j,desug}$ **then** $Prog^{A,\pi}_{out,t_j,desug}$

elseif ...

fi

ensuring $\bigwedge_j \left(P^{A,\pi}_{out,t_j,desug} \Rightarrow ensuring^{A,\pi}_{out,t_j,desug}\right)$

internal $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi})$ **where** $\bigvee_j P^{A,\pi}_{int,t_j,desug}$

Analogous to output.

...

Figure 12.5: Intermediate form of a desugared primitive automaton, with canonical action parameters and with all transition definitions for each kind of an action combined into a single transition definition

desugared and canonicalized as described in Sections 12.1 and 12.2. In Figure 12.5 and the discussion below, we indicate the syntactic forms that result from that desugaring by use of the *desug* subscript. We rely on the key semantic condition (mentioned in Section 11.4 and discussed in Section 12.6) that exactly one transition definition be defined for each assignment of values to $vars^{A,\pi}$ that satisfies $P^{A,\pi}_{kind}$. That is, within an automaton, all like-named transition definitions must have **where** clauses that are satisfiable only for disjoint sets of parameter values.[5]

First, notice that since all the contributing transition definitions are already desugared and canonicalized, each is is parameterized by $vars^{A,\pi}$. Hence, combining the parameters is trivial.

At first glance, combining local variables looks trickier. Each transition definition has local scope with respect to local variables. So, there may be any amount of duplication of variables among the sequences $localVars^{A,\pi}_{kind,t_j,desug}$. One might think that a correctly combined transition definition might need distinct local variables to store the values of the duplicate local variable appropriate to each contributing transition definition. However, for each assignment of values to $vars^{A,\pi}$ only one contributing transition definition can be defined for any assignment of values to its local variables. Therefore, there is at most one "useful" initial value for each local variable. Similarly, at most one contributing **eff** clause can make assignments to its local variables. Hence, duplicate declarations of local variables have no effect on the combined transition definition. Accordingly, we define $localVars^{A,\pi}$ to be the sequence of variables obtained by removing any duplicates from the concatenation of all sequences $localVars^{A,\pi}_{kind,t_j,desug}$.

In combining the various clauses of the contributing transitions, we use the **where** clauses of the contributing transitions as guards to select the correct case to use. The four clauses of the combined transition are combined as follows:

- The combined **where** clause is the disjunction of the **where** clauses from all the contributing transition definitions.

- For output and internal transition definitions, the combined **pre** clause checks that one set of parameters fulfills both the **where** and **pre** clauses of some contributing transition definition.

- The combined **eff** clause is a single **if**...**then**...**elseif**...**fi** statement in which the contributing **eff** clause is guarded by the associated **where** clause.

- Similarly, the combined **ensuring** clause asserts the appropriate contributing **ensuring** clause when the associated **where** clause is true. Note that since $P^{A,\pi}_{kind,t_j,desug}$ is defined on the initial values of $localVars^{A,\pi}_{kind,t_j,desug}$, assignments made to local variables in the **eff** clause have no effect on which **ensuring** clause is asserted.

[5]These semantic conditions also ensure that, in the absence of local variables, the resulting **where** clause can be eliminated because it will be equivalent to `true`.

```
automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[T]) where s = s2 ∪ {x} ∨ ¬(x ∈ s)
      eff if s = s2 ∪ {x} then seen[x] := true
          elseif ¬(x ∈ s) then seen[x] := false
          fi
    output found(x)
      pre seen[x]
```

Figure 12.6: Improved intermediate desugaring of the sample automaton `Watch`, obtained from the intermediate desugaring in Figure 12.4 by combining the transition definitions for `overflow`

$$
\begin{array}{l}
\textbf{automaton } A(\mathit{types}^A, \mathit{vars}^A) \\
\quad \ldots \\
\quad \textbf{states } \mathit{stateVars}^A := \mathit{initVals}^A \textbf{ initially } \sigma^A(P^A_{init}) \\
\quad \textbf{transitions} \\
\quad\quad \sigma^A \left[\sigma^{A,\pi}_{kind} \left[\begin{array}{l} \textbf{kind } \pi(\mathit{vars}^{A,\pi}; \textbf{ local } \mathit{localVars}^{A,\pi}) \textbf{ where } P^{A,\pi}_{kind,comb,t_1} \\ \quad \textbf{pre } \mathit{Pre}^{A,\pi}_{kind} \\ \quad \textbf{eff } \mathit{Prog}^{A,\pi}_{kind} \textbf{ ensuring } \mathit{ensuring}^{A,\pi}_{kind} \end{array} \right] \right] \\
\quad \ldots
\end{array}
$$

Figure 12.7: Final form of a desugared primitive automaton, with canonical action parameters, with all transition definitions for each kind of an action combined into a single transition definition, and with all variable references expanded.

Example 12.3 Consider the desugared and canonicalized automaton `Watch` shown in Figure 12.4. The only action with multiple transition definitions is the `overflow` input action. Following the above recipe, they are combined into the one equivalent action shown in Figure 12.6.

12.4 Combining aggregate sorts and expanding variable references

Section 11.2 described aggregate sorts that are automatically defined for the state and local variables of an automaton A (*i.e.*, $\mathit{States}[A, \mathit{types}^A]$ and $\mathit{Locals}[A, \mathit{types}^A, \mathit{kind}, \pi, t_j]$). Desugaring alters the automaton A and, consequently, can alter these aggregate sorts. In particular, as discussed in Section 12.3, combining multiple transition definitions for a particular action π in automaton A involves combining the local variables that appear in each transition into a single sequence. We collect together all the local variables for each action π of an automaton A into a single sequence of variables $\mathit{localVars}^{A,\pi}$, which is the concatenation (with duplicates removed) of the all sequences $\mathit{localVars}^{A,\pi}_{kind,t_j}$.

As a result, the aggregate sort for local variables also changes. Notationally, the *kind* and **case** labels t_j are dropped from the aggregate local sort name $Locals[A, types^A, kind, \pi, t_j]$. We define a new sort $Locals[A, types^A, \pi]$ for the combined transition definition to be a tuple with selection operators that are named, typed, and have values in accordance with the local variables in $localVars^{A,\pi}$. That is, the set of identifiers for the selection operators on the sort $Locals[A, types^A, \pi]$ is the union of the sets of identifiers for the selection operators on the sorts $Locals[A, types^A, kind, \pi, t_j]$. We change the sorts of the aggregate local and post-local variables A and A' to this new sort. This has the effect of collapsing multiple aggregate local and post-local variables each defined in the scope of one transition into a single local and post-local variable defined in all transitions for a given action.[6]).

Formally, for each transition definition t_j for a given *kind* of an action π in A, we define a resorting[7] that maps the aggregate local sort $Locals[A, types^A, kind, \pi, t_j]$ to the new aggregate local sort $Locals[A, types^A, \pi]$, and we apply that resorting to the transition definition before performing the combining step. As a result, each variable $A{:}Locals[A, types^A, kind, \pi, t_j]$ is mapped to a variable $A{:}Locals[A, types^A, \pi]$. Thus, local variable references using the notation $A.v$ form remain well defined and the resorting does not change the text of the transition definition. After combining, the sorts $Locals[A, types^A, kind, \pi, t_j]$ may be ignored.

In addition to introducing notations for aggregate local sorts, Section 11.2 also introduced notations for aggregate state sorts. These notations provided an additional, and potentially less ambiguous, way of referencing the values of local and state variables. We now desugar simple references to local and state variables to use the notations for aggregate local and state variables.

Formally, we define a substitution[8] σ^A to map state and post-state variables to terms. If x is a state variable or a post-state variable (*i.e.*, $x \in stateVars^A$ or $x \in postVars^A$), then $\sigma^A(x)) = A.x$, where A has sort $States[A, types^A]$ and the operator $\_\_.x$ has signature $States[A, types^A] \rightarrow T$, where T is the sort of x.

Similarly, for each transition definition π of type *kind*, we define a substitution $\sigma^{A,\pi}_{kind}$ to map local and post-local variables to terms. If x is a local or post-local variable (*i.e.*, $x \in localVars^{A,\pi}$ or $x \in localPostVars^{A_i,\pi}_{kind}$), then $\sigma^{A,\pi}_{kind}(x) = A.x$, where A has sort $Locals[A, types^A, kind, \pi]$, and the operator $\_\_.x$ has signature $Locals[A, types^A, kind, \pi] \rightarrow T$, where T is the sort of x.

Figure 12.7 shows the final form of a desugared primitive automaton with canonical action parameters and local variables and with all transition definitions for each kind of an action combined into a single transition definition, and with all variable references expanded. In that figure, we indicate the syntactic forms that result from the combining step by use of the *comb* subscript. Figure 12.8 shows the result of applying these substitutions to the sample primitive automata.

[6]To avoid complications that arise when new fields are added to an aggregate local tuple during the combining of local variables across transitions, we should disallow use of the constructor `[__,...]` for aggregate local sorts.

[7]See Chapter 17 for a formal definition of resortings, which map sorts to sorts.

[8]See Chapter 17 for a formal definition of substitutions, which map variables to terms.

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
    output receive(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
  states contents:Set[Msg] := {}
  transitions
    input send(n1, n2, m) where n1 = i ∧ n2 = j
      eff Channel.contents := insert(m, Channel.contents)
    output receive(n1, n2, m) where n1 = i ∧ n2 = j
      pre m ∈ Channel.contents
      eff Channel.contents := delete(m, Channel.contents)

automaton P(n:Int)
  signature
    input receive(i1, i2, x:Int) where i1 = n-1 ∧ i2 = n
    output send(i1, i2, x:Int) where i1 = n ∧ i2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(i1, i2, x) where i1 = n-1 ∧ i2 = n
      eff if P.val = 0 then P.val := x
          elseif x < P.val then
            P.toSend := insert(P.val, P.toSend);
            P.val := x
          elseif P.val < x then
            P.toSend := insert(x, P.toSend)
          fi
    output send(i1, i2, x) where i1 = n ∧ i2 = n+1
      pre x ∈ P.toSend
      eff P.toSend := delete(x, P.toSend)
    output overflow(i1, s; local t:Set[Int]) where i1 = n
      pre s = P.toSend ∧ n < size(s) ∧ P.t ⊆ s
      eff P.toSend := P.t

automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[T])
            where s = Watch.s2 ∪ {x} ∨ ¬(x ∈ s)
      eff
        if s = Watch.s2 ∪ {x} then Watch.seen[x] := true
        elseif ¬(x ∈ s) then Watch.seen[x] := false
        fi
    output found(x)
      pre Watch.seen[x]
```

Figure 12.8: Sample desugared automata `Channel`, `P`, and `Watch`, obtained from the intermediate desugarings in Figures 12.4 and 12.6 by desugaring references to state and local variables

12.5 Restrictions on the form of desugared automaton definitions

After the definition of a primitive automaton A has been desugared as described in Sections 12.1–12.4, it has the following properties.

- No **const** parameters appear in the signature of A.
- Each appearance of an action π in the signature of A is parameterized by the canonical action parameters $vars^{A,\pi}$ of π in A.
- Each transition definition of an action π is parameterized by the canonical action parameters $vars^{A,\pi}$ of π in A; *i.e.*, every parameter is a simple reference to a variable in $vars^{A,\pi}$.
- Each action name has at most one transition definition of each kind.
- Each reference to a state variable x of A, other than in the list of state variables in the **states** statement, has been replaced by the term $A.x$.
- Each reference to a post-state variable x' of A has been replaced by the term $A'.x$.
- Each reference to a local variable x in a transition of A, other than in the **local** clause of that transition definition, has been replaced by the term $A.x$.
- Each reference to a post-local variable x' in a transition of A has been replaced by the term $A'.x$.

12.6 Semantic proof obligations, revisited

We are now ready to formalize the semantic proof obligations for primitive automata introduced in Section 11.4. Previously, we said that for each action named π and each sequence of parameters values:

1. At most one of $P_{in}^{A,\pi}$, $P_{out}^{A,\pi}$, and $P_{int}^{A,\pi}$ is true.
2. If $P_{kind}^{A,\pi}$ is true, at least one $P_{kind,t_j}^{A,\pi}$ is true.
3. If $P_{kind}^{A,\pi}$ is true, at most one $P_{kind,t_j}^{A,\pi}$ is true

We explicitly did not define the phrase "sequence of parameters values" because these predicates may be stated in terms of different variables. In other words, $vars_{in}^{A,\pi}$ may be different from $vars_{out}^{A,\pi}$ and $vars_{in,t_1}^{A,\pi}$. Similarly, $vars_{in,t_1}^{A,\pi}$ may be different from $vars_{in,t_2}^{A,\pi}$. However, after desugaring and canonicalizing (but *before* combining), we have predicates that are semantically equivalent to those

in the original automaton, but defined over a common set of free variables. That is, all the free variables of all the predicates $\sigma^{A,\pi}_{kind}(P^{A,\pi}_{kind,desug})$ and $\sigma^{A,\pi}_{kind,t_j}(P^{A,\pi}_{kind,t_j,desug})$ are among $vars^A$ and $vars^{A,\pi}$.

The alert reader will realize that Tables 11.1 and 12.1 list $localVars^{A,\pi}_{kind,t_j}$ among the variables that may occur freely in $P^{A,\pi}_{kind,t_j}$ and $P^{A,\pi}_{kind,t_j,desug}$ and might therefore conclude that the aforementioned predicates are not "defined over a common set of free variables". However, as noted Section 11.2, a transition π is defined only for values of its parameters that, together with *some* choice of initial values for its local variables, satisfy the **where** clause of the transition definition. Thus, for the purposes of formalizing the semantic proof obligations for transition definitions, local variables should be existentially bound, not free in **where** clauses, that is, $P^{A,\pi}_{kind,t_j,desug}$ should be preceded by $\exists localVars^{A,\pi}_{kind,t_j}$.

The semantic proof obligations we introduced in Section 11.4 can be stated precisely as follows. We require that for each action name π, all values of $vars^A$, and all values of $vars^{A,\pi}$, the following statements must be provable from the axioms provided by IOA's built-in types, by the theories associated with the type definitions and the **axioms** in the IOA specification that contains the automaton definition, and by the theories associated with the **assumes** clause of that definition.

✓ $$\neg\left(\sigma^{A,\pi}_{in}(P^{A,\pi}_{in,desug}) \wedge \sigma^{A,\pi}_{out}(P^{A,\pi}_{out,desug})\right), \tag{12.1}$$

✓ $$\neg\left(\sigma^{A,\pi}_{in}(P^{A,\pi}_{in,desug}) \wedge \sigma^{A,\pi}_{int}(P^{A,\pi}_{int,desug})\right), \tag{12.2}$$

✓ $$\neg\left(\sigma^{A,\pi}_{out}(P^{A,\pi}_{out,desug}) \wedge \sigma^{A,\pi}_{int}(P^{A,\pi}_{int,desug})\right), \tag{12.3}$$

✓ $$\sigma^{A,\pi}_{kind}(P^{A,\pi}_{kind,desug}) \Rightarrow \bigvee_j \exists localVars^{A,\pi}_{kind,t_j}\sigma^{A,\pi}_{kind,t_j}(P^{A,\pi}_{kind,t_j,desug}), \text{ and} \tag{12.4}$$

✓ $$\sigma^{A,\pi}_{kind}(P^{A,\pi}_{kind,desug}) \Rightarrow \tag{12.5}$$
$$\neg\left(\exists localVars^{A,\pi}_{kind,t_j}\sigma^{A,\pi}_{kind,t_j}(P^{A,\pi}_{kind,t_j,desug}) \wedge \exists localVars^{A,\pi}_{kind,t_k}\sigma^{A,\pi}_{kind,t_k}(P^{A,\pi}_{kind,t_k,desug})\right),$$
when $j \neq k$.

Chapter 13

Definitions for composite automata

Today we have naming of parts. Yesterday,
We had daily cleaning. And tomorrow morning,
We shall have what to do after firing. But today,
Today we have naming of parts. Japonica
Glistens like coral in all of the neighboring gardens,
And today we have naming of parts.

— Henry Reed [103]

This chapter introduces notations and semantic checks for composite IOA automata. Section 13.1 describes the syntactic structures that may appear in an IOA description of a composite I/O automaton. Section 13.2 describes notations for the state variables of a composite automaton. When component automata have type parameters, the sorts of these state variables are obtained by mapping the formal type parameters of the component automata to the actual parameters used to instantiate those components in the composition. Finally, Sections 13.3 and 13.4 describe the conditions that descriptions of composite automata must satisfy to be semantically valid.

13.1 Syntax

As for primitive automata, we introduce a labeling of the syntactic elements of composite IOA programs in order to facilitate describing their syntactic manipulation. Figure 13.1 indicates a particular labeling of the expressions that can appear in the IOA definition of a composite I/O automaton. Again, we have selected the granularity of this labeling to expose just those elements of composite IOA programs that are needed in Chapter 15 to describe the expansion of composite

SYNTACTIC STRUCTURE	FREE VARIABLES
$actuals^{D,C_i}$	$vars^D$, $vars^{D,C_i}$
P^{D,C_i}	$vars^D$, $vars^{D,C_i}$
$H^{D,\pi_p}_{hide_p}$	$vars^D$, $vars^{D,\pi_p}_{hide_p}$
$params^{D,\pi_p}_{hide_p}$	$vars^D$, $vars^{D,\pi_p}_{hide_p}$
Inv^D_x	$vars^D$, $stateVars^D$

Table 13.1: Variables that can occur freely in terms in the definition of a composite automaton. Variables listed on the right may occur freely in the syntactic structure listed to their left.

automata into primitive form.

automaton $D(types^D, vars^D)$
  assumes $Assumptions$
  components
    $C_1[vars^{D,C_1}] : A_1(actualTypes^{D,C_1}, actuals^{D,C_1})$ **where** P^{D,C_1};
    ...;
    $C_n[vars^{D,C_n}] : A_n(actualTypes^{D,C_n}, actuals^{D,C_n})$ **where** P^{D,C_n}
  hidden
    $\pi_1(params^{D,\pi_1}_{hide_1})$ **where** $H^{D,\pi_1}_{hide_1}$;
    ...;
    $\pi_m(params^{D,\pi_m}_{hide_m})$ **where** $H^{D,\pi_m}_{hide_m}$

invariant of D : Inv^D_1; ... Inv^D_z

Figure 13.1: General form of a composite automaton

In Figure 13.1, parameterized components named $C_1, \ldots, C_n$ are based on instantiations of automata named $A_1, \ldots, A_n$. The formal parameters of component C_i are $vars^{D,C_i}$, and the actual parameters of automaton A_i consist of a sequence $actualTypes^{D,C_i}$ of sorts and a sequence $actuals^{D,C_i}$ of terms. IOA permits the specification of C_i to be abbreviated by deleting the colon and the following expression when C_i and A_i are named by the same identifier, $actualTypes^{D,C_i}$ is empty, and $actuals^{D,C_i} = vars^{D,C_i}$ (*e.g.*, see component `P` in Example 10.4). In the specification of **hidden** actions, $params^{D,\pi_p}_{hide_p}$ is a sequence of terms, analogous to $params^{A,\pi_p}_{out,t_1}$, and we define $vars^{D,\pi_p}_{hide_p}$ to be the set of variables that occur freely in $params^{D,\pi_p}_{hide_p}$ but are not in $vars^D$. Each **invariant** of D is stated as a predicate Inv^D_x.

Example 10.4 conforms to this general form, as follows.

- The first component of `Sys` is named `C`. Its parameters, $vars^{\texttt{Sys,C}}$, are $\langle$`n:Int`$\rangle$, and it is based on the automaton `Channel`, for which it supplies the actual parameters $actualTypes^{\texttt{Sys,C}} = \langle$`Int, Int`$\rangle$ and $actuals^{\texttt{Sys,C}} = \langle$`n, n+1`$\rangle$.

- The second component of `Sys` is named `P`. It has the same parameters as `C`. By the conventions for abbreviating component descriptions, it is based on the automaton of the same name, for which it supplies the actual parameters $actuals^{\texttt{Sys,P}} = \langle$`n`$\rangle$; in this case, $actualTypes^{\texttt{Sys,P}}$ is empty (as required to use this abbreviated form).

- The third component of `Sys`, named `W`, has no parameters. It is based on the automaton `Watch`, for which it supplies the actual parameters $actualTypes^{\texttt{Sys,W}} = \langle$`Int`$\rangle$ and $actuals^{\texttt{Sys,W}} = \langle$`between(1,nProcesses)`$\rangle$.

- The `send` actions that `Sys` inherits from `P[nProcesses]` are hidden as internal actions in `Sys`. The parameters $params_{hide_1}^{\texttt{Sys,send}} = \langle$`nProcesses, nProcesses+1, m`$\rangle$ in the single clause in the **hidden** statement involve a single free variable in $vars_{hide_1}^{\texttt{Sys,send}} = \langle$`m:Int`$\rangle$, and $H_{hide_1}^{\texttt{Sys,send}}$ is *true*.

- The predicate
  `∀ m:Int ∀ n:Int (1 ≤ m ∧ m < n ∧ n ≤ nProcesses`
  `⇒ P[m].val < P[n].val ∨ P[n].val = 0)`
  is invariant $Inv_1^{\texttt{Sys}}$ of `Sys`.

13.2 State variables of composite automata

The definition of a composite automaton in IOA does not mention the automaton's state variables explicitly. Rather, its **components** statement implicitly introduces a single state variable for each component. We first describe the notations IOA provides for state variables associated with component automata that have no type parameters. Then we describe how these notations extend to state variables associated with component automata that have type parameters. Our goal is to provide a precise explanation of notations for state variables such as `P[m].val`, which appears in the invariant for the sample composite automaton `Sys`.

As for primitive automata (see Section 11.2), we automatically define a sort $States[D, types^D]$ representing the aggregate states of a composite automaton D, and we also define aggregate state and post-state variables D and D' of sort $States[D, types^D]$. Similarly, we treat the sort $States[D, types^D]$ in the same fashion as for primitive automata, namely, as a tuple of state variables: we define the aggregate state of a composite automaton D to be a tuple containing a state variable for each component automaton, and we use the names of the components (*i.e.*, $C_1, \ldots, C_n$) as the names of these state variables and of the corresponding selectors (*i.e.*, $\_\_.C_1, \ldots, \_\_.C_n$) of $States[D, types^D]$.

13.2.1 State variables for components with no type parameters

Defining the sort of the state variable C_i is simplest when the component C_i does not have parameters and when the automaton A_i on which C_i is based does not have type parameters. For each such component C_i, the state variable C_i of D has sort $States[A_i]$, and the selector $\_\_.C_i$ has signature $States[D, types^D] \rightarrow States[A_i]$.

When the component C_i has parameters, but A_i still does not have type parameters, the situation is slightly more complicated, because the composite automaton D may contain multiple instances of A_i. For example, the composite automaton `Sys` contains `nProcesses` instances of the component automaton `P`, each with its own state variables `val` and `toSend`. These instances are parameterized by a single integer `n` and are distinguished by the component names `P[1]`, ..., `P[nProcesses]`.

For each parameterized component C_i, the corresponding state variable C_i does not refer to the aggregate state of a single instance of A_i. Rather, it refers to a *map* from the values of the parameters $vars^{D,C_i}$ of C_i to the aggregate states of A_i. That is, the state variable C_i has sort `Map[`$types^{D,C_i}$`,` $States[A_i]$`]`, where $types^{D,C_i}$ is the sequence of sorts of the variables in $vars^{D,C_i}$. The selection operator $\_\_.C_i$ has signature $States[D, types^D] \rightarrow$ `Map[`$types^{D,C_i}$`,` $States[A_i]$`]`.

For example, the state variable `P` of `Sys` has sort `Map[Int,States[P]]`. Hence, `P[n]` is a legitimate term with sort `States[P]`, and the term `P[n].val` has sort `Int`. Likewise, the selection operator `__.P` has signature `States[Sys]` $\rightarrow$ `Map[Int,States[P]]`, and `Sys.P[n].val` is an alternative notation for the state variable `val` that `Sys` inherits from component `P[n]`.

13.2.2 Resortings for automata with type parameters

Defining the sort of the state variable C_i is more complicated when A_i has type parameters. Since the semantics for IOA are defined using multisorted, first-order logic, we cannot quantify over sorts or use sorts as component indices. Instead, different instances of A_i, corresponding to different actual types, must be described in separate clauses in the **components** statement, where they are further distinguished by different component names. As a result, there can be only finitely many differently typed instantiations of A_i, even though altogether there may be infinitely many instances of A_i that are distinguished by the values of their non-type parameters. For example, a composite automaton might contain channel components that transmit finitely many different types of messages, but there may be infinitely many instances of such a component that transmits a given type of message.

When a component C_i is based on an automaton A_i parameterized by the sorts $types^{A_i}$, we define a resorting ρ_i (which we write as ρ^{C_i} in contexts, such as $\rho^{\mathtt{W}}$, where it is more convenient to use the name of the component rather than its position in the list of all components) that maps $types^{A_i}$ to $actualTypes^{D,C_i}$. For example, $\rho^{\mathtt{W}}$ maps $types^{\mathtt{Watch}} = \langle \mathtt{T} \rangle$ to $actualTypes^{\mathtt{Sys,W}} = \langle \mathtt{Int} \rangle$, and $\rho^{\mathtt{C}}$ maps $types^{\mathtt{Channel}} = \langle \mathtt{Node,\ Msg} \rangle$ to $actualTypes^{\mathtt{Sys,C}} = \langle \mathtt{Int,\ Int} \rangle$.

As described in Section 17, there is a natural way to extend the resorting ρ_i to map arbitrary sorts involving the formal type parameters in the defining automaton A_i to sorts involving the corresponding actual types that the component C_i supplies for A_i. For example, this extension maps the automatically defined sort $States[A_i, types^{A_i}]$ for the state of A_i to the sort $States[A_i, actualTypes^{D,C_i}]$ for the state of the instances of A_i corresponding to the component C_i.[1]

The resorting ρ_i also extends naturally to map operators with signatures involving the formal type parameters in the defining automaton A_i to operators with signatures involving the corresponding actual types that the component C_i supplies for A_i. Thus, for example, ρ^C maps

```
States[Channel,Node,Msg] = tuple of contents: Set[Msg]
```

to

```
States[Channel,Int,Int] = tuple of contents: Set[Int]
```

and it maps the signature of the selection operator `__.contents` from `States[Channel,Node,Msg]` → `Set[Msg]` to `States[Channel,Int,Int]` → `Set[Int]`.

13.2.3 State variables for components with type parameters

When A_i has type parameters, we employ a resorting of its aggregate state sort to define the sort of the state variable C_i of D. In the simple case when the component C_i does not have any parameters, the state variable C_i has sort $States[A_i, actualTypes^{D,C_i}]$, and the selection operator $\_\_.C_i$ has signature $States[D, types^D] \rightarrow States[A_i, actualTypes^{D,C_i}]$.

For example, the state variable `W` of `Sys` has sort `States[Watch,Int]`, the term `W.seen` has sort `Array[Int,Bool]`, the selection operator `__.W` has signature `States[Sys]` → `States[Watch,Int]`, and `Sys.Watch.seen` is an alternative notation for the state variable `seen` that `Sys` inherits from component `W`.

In the case when the component C_i has parameters (and the automaton A_i has type parameters), the state variable C_i has sort `Map[`$types^{D,C_i}$`,` $States[A_i, actualTypes^{D,C_i}]$`]`, where $types^{D,C_i}$ is the sequence of sorts of the variables in $vars^{D,C_i}$, and the selection operator $\_\_.C_i$ has signature $States[D, types^D] \rightarrow$ `Map[`$types^{D,C_i}$`,` $States[A_i, actualTypes^{D,C_i}]$`]`.

For example, the state variable `C` of `Sys` has sort `Map[Int,States[Channel,Int,Int]]`, the term `C[n]` has sort `States[Channel[Int,Int]`, the term `C[n].contents` has sort `Set[Int]`, the selection operator `__.C` has signature `States[Sys]` → `Map[Int,States[Channel,Int,Int]`, and `C[n].contents` is an alternative notation for the state variable `contents` that `Sys` inherits from component `C[n]`.

[1] Although A_i, $types^A$, C_i, and $actualTypes^{D,C_i}$ appear as subsorts of a sort constructor `States[__,...]`, IOA assigns no semantics to these sorts. Syntactically, however, they are treated in the same fashion as other sorts; in particular, the resorting ρ_i replaces $types^{A_i}$ by $actualTypes^{D,C_i}$.

13.3 Static semantic checks

The following must be true for an IOA program to represent a valid composite I/O automaton and can be checked statically. These checks are currently performed by `ioaCheck`, the IOA parser and static-semantic checker.

- ✓ No sort appears more than once in $types^D$.
- ✓ Each component name (*i.e.*, C_i) occurs at most once.
- ✓ The sequences $vars^D$ and $vars^{D,C_i}$ of variables contain no duplicates; furthermore, no variable appears in both $vars^D$ and $vars^{D,C_i}$ for any value of i.
- ✓ Each component automaton is supplied with the appropriate number of actual types, that is, $actualTypes^{D,C_i}$ has the same length as $types^{A_i}$.
- ✓ For every operator f in a theory specified in the **assumes** clause of the automaton A_i, a corresponding operator $\rho_i(f)$ must be introduced by a type definition or **axioms** clause in the IOA specification that contains the definition of D, by a theory specified in the **assumes** clause of D, or by a built-in datatype of IOA.
- ✓ Each component automaton is supplied with the appropriate number and sorts of its other actual parameters, that is, $actuals^{D,C_i}$ has the same length as $vars^{A_i}$ and the same sorts as $\rho_i(vars^{A_i})$.
- ✓ Each component automaton is supplied with actual types that do not reduce the number of distinct state variables. That is, all selectors of $States[A_i, actualTypes^{D,C_i}]$ are distinct.
- ✓ All occurrences of an action name π in all component automata have the same number and sorts of parameters; that is, if π is an action name in both A_i and A_j, then $vars^{A_i,\pi}$ has the same length as $vars^{A_j,\pi}$, and $\rho_i(vars^{\hat{A}i,\pi})$ has the same sort as $\rho_j(vars^{\hat{A}j,\pi})$.
- ✓ Each action name in a **hidden** statement must be an action name in some component automaton.
- ✓ All occurrences of an action name π in a **hidden** statement have the same number and sorts of parameters as the occurrences of the action name π in the component automata; that is, if π is an action name in some A_i and $\pi = \pi_p$ for the **hidden** clause p, then $vars^{A_i,\pi}$ has the same length as $params^{D,\pi_p}_{hide_p}$, and $\rho_i(vars^{\hat{A}i,\pi})$ has the same sorts as $params^{D,\pi_p}_{hide_p}$.
- ✓ Any variable that occurs freely in a term used as a parameter or predicate, in the definition of a composite automaton must satisfy the restrictions imposed by Table 13.1.

13.4 Semantic proof obligations

The following must also be true for an IOA program to represent a valid I/O automaton. Except in special cases, these conditions cannot be checked automatically, because they may require nontrivial proofs (or even be undecidable); hence static semantic checkers must translate all but the simplest of them into proof obligations for an automated proof assistant. [2]

- ✓ Only output actions may be hidden.
- ✓ The components of a composite automaton must have disjoint sets of output actions.
- ✓ The set of internal actions for any component must be disjoint from the set of all actions of every other component.

We will express these these proof obligations in first-order logic in Section 15.4 using syntactic forms we define earlier in Chapter 15.

[2] An implementation of these checks might reduce the number of errors reported by first confirming that the composition contains no duplicate instances of any component automaton that contains internal or output actions. Any such duplication would necessarily cause violations of the latter two checks.

Chapter 14

Expanding component automata

To keep every cog and wheel is the first precaution of intelligent tinkering. — Leopold, Aldo [77]

Before we can describe the contribution of a component C_i of a composite automaton D to the expansion of D into a primitive automaton *DExpanded*, we must take four preparatory steps. The result is a component that represents the instantiation of automaton A_i on which C_i is based using the actual parameters supplied by the component and whose variables have been translated into a unified name space used for *DExpanded*.

The first step is to desugar the definition of each component automaton A_i as described in Chapter 12. In the discussion below, we refer to this desugared version of A_i as $\hat{A}_i$ and assume that it satisfies the restrictions listed in Section 12.5. The second step, shown in Section 14.1, is to replace, throughout the entire definition of the automaton $\hat{A}_i$, the formal type parameters $types^{\hat{A}_i}$ of $\hat{A}_i$ by the actual types $actualTypes^{D,C_i}$ supplied by the component C_i. The third step is to replace the formal automaton (non-type) parameters $vars^{A_i}$ by the actual parameters $actuals^{D,C_i}$ supplied by the component C_i. The fourth step is to translate the aggregate state variables, aggregate local variables, and action parameters from the name space of $\hat{A}_i$ into a unified name space for *DExpanded*. (It is not necessary to translate individual state and local variables, because references to them have been eliminated by the desugaring described in Section 12.4.) Sections 14.2 describes how we choose canonical action parameters for the unified name space. Section 14.3 describes the substitution we use to perform both this translation and the instantiation of actual automaton parameters for the previous step. Table 14.8 summarizes the notation, figures, and examples we use to present these stages.

Section 14.4 describes the result of applying these replacements and translations to individual component automata. It sets the stage Chapter 15, which describes how to combine the expanded

RESORTING	DOMAIN	RANGE
$\rho^{\mathtt{C}}$	`Node`	`Int`
	`Msg`	`Int`
	`Set[Msg]`	`Set[Int]`
	`States[Channel,Node,Msg]`	`States[Channel,Int,Int]`
$\rho^{\mathtt{W}}$	`T`	`Int`
	`Set[T]`	`Set[Int]`
	`Array[T,Bool]`	`Array[Int,Bool]`
	`States[Watch,T]`	`States[Watch,Int]`
	`Locals[Watch,T,overflow]`	`Locals[Watch,Int,overflow]`

Table 14.1: Mappings of sorts by resortings in the composite automaton `Sys`. Resortings listed on the left map domain sorts to their right to the range sorts on their far right.

components into a description of *DExpanded* by developing explicit representations for its signature and transition definitions.

14.1 Resorting component automata

We produce a definition of the instances of $\hat{A}_i$ whose sorts correspond to those of the component C_i by replacing the formal type parameters $types^{\hat{A}_i}$ of $\hat{A}_i$ with the actual types $actualTypes^{D,C_i}$ supplied by the component C_i. This replacement is accomplished by applying the resorting ρ_i, defined in Section 13.2 to the entire definition of the automaton $\hat{A}_i$. The precise definition of resortings and a full description of how resortings are extended to perform this replacement throughout the entire definition of the automaton $\hat{A}_i$ are given in Chapter 17. We denote the resulting definition by $\rho_i\hat{A}_i$.

Example 14.1 Tables 14.1–14.3 show how the resortings $\rho^{\mathtt{C}}$ and $\rho^{\mathtt{W}}$, induced by the **components** statement of the sample automaton `Sys` in Example 10.4, map the sorts, variables, and operators of the component automata.[1] The resorted components $\rho^{\mathtt{C}}\mathtt{Channel}$ and $\rho^{\mathtt{W}}\mathtt{Watch}$ of the composite automaton `Sys` are shown in Figure 14.1. Since the component automaton `P` of `Sys` does not have any type parameters, $\rho^{\mathtt{P}}$ is the identity, and the resorted component $\rho^{\mathtt{P}}\mathtt{P}$ is the same as shown in Figure 12.8.

[1]The table shows only the non-identity mappings of sorts, variables, and operators. Sorts, variables, and operators that appear in the sample automata, but are not shown in the table, are mapped to themselves.

RESORTING	DOMAIN	RANGE
$\rho^{\mathtt{C}}$	i:Node	i:Int
	j:Node	j:Int
	contents:Set[Msg]	contents:Set[Int]
	n1:Node	n1:Int
	n2:Node	n2:Int
	m:Msg	m:Int
$\rho^{\mathtt{W}}$	what:Set[T]	what:Set[Int]
	seen:Array[T,Bool]	seen:Array[Int,Bool]
	x:T	x:Int
	x:T	x:Int
	s:Set[T]	s:Set[Int]
	s2:Set[T]	s2:Set[Int]

Table 14.2: Mappings of variables by resortings in the composite automaton `Sys`. Resortings listed on the left map domain variables to their right to the range variables on their far right.

```
% Resorting of Channel for component C of Sys
automaton Channel(Node, Msg:type, i, j:Int)
  signature
    input send(n1, n2:Int, m:Int) where n1 = i ∧ n2 = j
    output receive(n1, n2:Int, m:Int) where n1 = i ∧ n2 = j
  states contents:Set[Int] := {}
  transitions
    input send(n1, n2, m) where n1 = i ∧ n2 = j
      eff Channel.contents := insert(m, Channel.contents)
    output receive(n1, n2, m) where n1 = i ∧ n2 = j
      pre m ∈ Channel.contents
      eff Channel.contents := delete(m, Channel.contents)

% Resorting of Watch for component W of Sys
automaton Watch(T:Type, what:Set[Int])
  signature
    input overflow(x:Int, s:Set[Int]) where x ∈ what
    output found(x:Int) where x ∈ what
  states seen:Array[Int,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[Int])
            where s = Watch.s2 ∪ {x} ∨ ¬(x ∈ s)
      eff if s = Watch.s2 ∪ {x} then Watch.seen[x] := true
          elseif ¬(x ∈ s) then Watch.seen[x] := false
          fi
    output found(x)
      pre Watch.seen[x]
```

Figure 14.1: Sample component automata `Channel` and `Watch`, obtained by resorting the desugared automata shown in Figure 12.8

RESORTING	OPERATOR	ORIGINAL AND NEW SIGNATURES
$\rho^{\mathtt{C}}$	=	Node,Node→Bool Int,Int→Bool
	=	Msg,Msg→Bool Int,Int→Bool
		→Set[Msg] →Set[Int]
	∈	Msg,Set[Msg]→Bool Int,Set[Int]→Bool
	insert	Msg,Set[Msg]→Set[Msg] Int,Set[Int]→Set[Int]
	delete	Msg,Set[Msg]→Set[Msg] Int,Set[Int]→Set[Int]
	__.contents	States[Channel,Node,Msg]→Set[Msg] States[Channel,Int,Int]→Set[Int]
$\rho^{\mathtt{W}}$	[__]	T→Bool Int→Bool
	{__}	T→Set[T] Int→Set[Int]
	=	Set[T],Set[T]→Bool Set[Int],Set[Int]→Bool
	∈	T,Set[T]→Bool Int,Set[Int]→Bool
	∪	Set[T],Set[T]→Set[T] Set[Int],Set[Int]→Set[Int]
	__.seen	States[Watch,T]→Array[T,Bool] States[Watch,Int]→Array[Int,Bool]
	__.s2	Locals[Watch,T,overflow]→Set[T] Locals[Watch,Int,overflow]→Set[Int]

Table 14.3: Mappings of operators by resortings in the composite automaton Sys. Resortings listed on the left map domain operators to their right to the range operators on their far right.

14.2 Introducing canonical names for parameters

For each action name π in some component C_i of D, we pick a sequence $vars^{D,\pi}$ of variables to be the *canonical action parameters* of π in D. Since the static checks ensure the number and sorts of variables in $\rho_i(vars^{\hat{A}i,\pi})$ are the same for all components C_i, we take $vars^{D,\pi}$ to be $\rho_i(vars^{\hat{A}i,\pi})$ for the smallest i such that π is the name of an action in C_i and this choice does not cause variables to clash. In particular, no variable in $vars^{D,\pi}$ should be a parameter of D (*i.e.*, $vars^{D,\pi}$ and $vars^{D}$ should be disjoint) nor of any component C_i (*i.e.*, $vars^{D,\pi}$ and $vars^{D,C_i}$ should be disjoint).[2]

If $vars^{D,\pi}$ cannot be defined in this fashion (without causing variables to clash), then we let i be the smallest integer such that π is the name of an action in C_i, and we take $vars^{D,\pi}$ to be $\rho_i(vars^{\hat{A}i,\pi})$ with any clashing variables replaced by fresh variables, that is, with variables not in $vars^{D}$ nor any $vars^{D,C_i}$.

14.3 Substitutions

For each component C_i of a composite automaton D, we define a substitution σ_i (which we write as σ^{C_i} in contexts, such as $\sigma^{\mathtt{W}}$, where it is more convenient to use the name of the component rather than its position in the list of all components) to map the non-type parameters $vars^{\rho_i\hat{A}_i} = \rho_i(vars^{\hat{A}_i})$ of the component automaton $\rho_i\hat{A}_i$ to the corresponding actual parameters $actuals^{D,C_i}$ and to map the aggregate state and post-state variables of $\rho_i\hat{A}_i$ to the appropriate state components in the composite automaton. For each action π of C_i, we also define a substitution $\sigma_{i,\pi}$ to be the same as σ_i, except that it also maps the canonical action parameters $vars^{\rho_i\hat{A}_i,\pi} = \rho_i(vars^{\hat{A}_i})$ of $\rho_i\hat{A}_i$ to the corresponding canonical action parameters $vars^{D,\pi}$ in D, and that it maps the aggregate local and post-local variables for transition definitions in $\rho_i\hat{A}_i$ to the appropriate local and post-local values in the composite automaton.

These substitutions[3] are summarized in Table 14.4 and defined by rules 1–9 below.

1. If x is a non-type parameter of $\hat{A}_i$ (*i.e.*, $x \in vars^{\rho_i\hat{A}_i}$), then $\sigma_i\rho_i(x)$ is the corresponding element of $actuals^{D,C_i}$.

2. If C_i has no parameters and x is the variable A_i of sort $States[A_i, actualTypes^{D,C_i}]$ representing the aggregate states of $\rho_i\hat{A}_i$, then $\sigma_i(x)$ is the state variable for the component C_i of D, which has the same sort as A_i.

3. If C_i has parameters and x is the variable A_i of sort $States[A_i, actualTypes^{D,C_i}]$, then $\sigma_i(x)$ is the term $C_i[vars^{D,C_i}]$, where C_i is the state variable for the component C_i of D, which has

[2] It is not necessary to avoid clashes with the state variables $\rho_i(stateVars^{A_i})$ or post-state variables $\rho_i(postVars^{A_i})$ of C_i, because desugaring has replaced references to such variables x by terms $C_i.x$.

[3] See Chapter 17 for a precise definition of substitutions, which ensures that they do not capture **local**, **for**, **choose**, or quantified variables.

SUBSTITUTION	DOMAIN	RANGE	RULE
σ_i	$vars^{\rho_i \hat{A}_i}$	$actuals^{D,C_i}$	rule 1
	A_i:$States[A_i, actualTypes^{D,C_i}]$	C_i	rule 2
	A_i:$States[A_i, actualTypes^{D,C_i}]$	$C_i[vars^{D,C_i}]$	rule 3
$\sigma_{i,\pi}$	$vars^{\rho_i \hat{A}_i}$	$actuals^{D,C_i}$	rule 1
	A_i:$States[A_i, actualTypes^{D,C_i}]$	C_i	rule 2
	A_i:$States[A_i, actualTypes^{D,C_i}]$	$C_i[vars^{D,C_i}]$	rule 3
	A'_i:$States[A_i, actualTypes^{D,C_i}]$	C'_i	rule 4
	A'_i:$States[A_i, actualTypes^{D,C_i}]$	$C'_i[vars^{D,C_i}]$	rule 5
	$vars^{\rho_i \hat{A}_i,\pi}$	$vars^{D,\pi}$	rule 7
	A_i:$Locals[A_i, actualTypes^{D,C_i}, \pi]$	C_i	rule 8
	A'_i:$Locals[A_i, actualTypes^{D,C_i}, \pi]$	C'_i	rule 8
	A_i:$Locals[A_i, actualTypes^{D,C_i}, \pi]$	$C_i[vars^{D,C_i}]$	rule 9
	A'_i:$Locals[A_i, actualTypes^{D,C_i}, \pi]$	$C'_i[vars^{D,C_i}]$	rule 9

Table 14.4: Substitutions used in canonicalizing component automata. Substitutions listed on the left map variables in the domains to their right to range variables according to the listed rules.

sort `Map[`$types^{D,C_i}$, $States[A_i, actualTypes^{D,C_i}]$`]`.

4. If C_i has no parameters and x is the variable A'_i of sort $States[A_i, actualTypes^{D,C_i}]$ representing the aggregate post-states of $\rho_i \hat{A}_i$, then $\sigma_i(x)$ is the post-state variable C'_i for the component C_i of D.

5. If C_i has parameters and x is the variable A'_i of sort $States[A_i, actualTypes^{D,C_i}]$, then $\sigma_i(x)$ is the term $C'_i[vars^{D,C_i}]$, where C'_i is the post-state variable for the component C_i of D, which has sort `Map[`$types^{D,C_i}$, $States[A_i, actualTypes^{D,C_i}]$`]`.

6. There is no rule 6! [20]

7. If x is a canonical action parameter (*i.e.*, $x \in vars^{\hat{A}_i,\pi}$), then $\sigma_{i,\pi}\rho_i(x)$ is the corresponding element of $vars^{D,\pi}$.

8. If C_i has no parameters and x is the variable A_i of sort $Locals[A_i, actualTypes^{D,C_i,\pi}]$ (or the variable A'_i of the same sort) representing the aggregate local (or post-local) variables for a transition definition, then $\sigma_i(x)$ is the local variable C_i (or the post-local variable C'_i) for the transition definition in D, which has the same sort as A_i.

9. If C_i has parameters and x is the variable A_i of sort $Locals[A_i, actualTypes^{D,C_i,\pi}]$ (or the variable A'_i of the same sort), then $\sigma_i(x)$ is the term $C_i[vars^{D,C_i}]$ (or the term $C'_i[vars^{D,C_i}]$),

where C_i and C_i' are the aggregate local and post-local variables in D, which have sort $\texttt{Map}[types^{D,C_i}, Locals[A_i, actualTypes^{D,C_i}, \pi]]$.

14.4 Canonical component automata

For each component C_i of D, we obtain a canonical automaton definition C_i for that component by applying ρ_i and then σ_i to the desugared definition $\hat{A}_i$ of A_i. Figure 14.2 shows the general form for such canonical component automata.

In the list of parameters for C_i, the type parameters $types^D$ of D replace the type parameters $types^{\hat{A}_i}$ of $\hat{A}_i$, and the variables $vars^D$ and $vars^{D,C_i}$ that parameterize D and its component C_i replace the individual parameters $vars^{A_i}$ of $\hat{A}_i$. The body of the automaton definition for C_i is obtained by applying the resorting ρ_i to the body of the automaton definition for $\hat{A}_i$, thereby eliminating all references to the type parameters in $types^{\hat{A}_i}$, to obtain a resorted definition for an automaton $\rho_i\hat{A}_i$ and then by applying the substitution σ_i to this resorted definition, thereby eliminating all references to the individual parameters in $vars^{A_i}$. We do not apply σ_i to $stateVars^{\rho_i A_i}$, because we wish to preserve the names of the state variables in $stateVars^{A_i}$. No ambiguity arises, because the desugaring described in Section 12.4 has replaced all references to state variables x in the definition of $\hat{A}_i$ with terms of the form $A_i.x$. For each action π, we also apply $\sigma_{i,\pi}$ to the **where** clause $P_{kind}^{\rho_i\hat{A}_i,\pi}$ for π in the signature of $\rho_i\hat{A}_i$ and to the transition definition for π in $\rho_i\hat{A}_i$.

Despite the absence of ambiguity, the automaton C_i may not pass the static semantic requirements in Section 11.3 that prohibit any clashes between state variables and automaton parameters. Furthermore, if C_i has non-type parameters, the aggregate state variable for the automaton is a map as specified in Section 13.2 rather than a tuple as specified for primitive automata in Section 11.2.

Table 14.8 shows the steps taken to expand canonical component automata. The "Original" column lists the names for syntactic elements of automata introduced in Chapter 11. The notation given in the "Desugared" column describes the result of desugaring such automata as described in Chapter 12. The elements listed in the the "Resorted" column result from the resorting of desugared component automata that Section 14.1 describes. Syntactic elements listed in the "Expanded" column are derived in Section 14.3 from resorted automata. Finally, names that appear in the "Component" column are just synonyms for the values in the previous column. We use these simpler synonyms in Chapter 15.

Example 14.2 We derive the component automata `C`, `P`, and `W` of the composite automaton `Sys` by applying the substitutions shown in Tables 14.5–14.7 to the resorted automata $\rho^{\texttt{C}}\texttt{Channel}$ and $\rho^{\texttt{W}}\texttt{Watch}$ shown in Figure 14.1 and to the canonicalized automaton `P` shown in Figure 12.8. Since the per-action substitutions (*e.g.*, $\sigma^{\texttt{C},\texttt{send}}$) are always extensions of the per-component substitutions (*e.g.*, $\sigma^{\texttt{C}}$), these tables show only the *additional* mappings that distinguish the per-action substi-

automaton $C_i(types^D, vars^D, vars^{D,C_i})$

signature

kind $\pi(vars^{D,\pi})$ **where** $\sigma_{i,\pi}(P^{\rho_i \hat{A}_i,\pi}_{kind})$

...

states $stateVars^{\rho_i A_i} := \sigma_i(initVals^{\rho_i \hat{A}_i})$ **initially** $\sigma_i(P^{\rho_i \hat{A}_i}_{init})$

transitions

$$\sigma_{i,\pi}\left[\begin{array}{l} \textbf{kind } \pi(vars^{\rho_i \hat{A}_i,\pi};\ \textbf{local } localVars^{\rho_i \hat{A}_i,\pi}) \textbf{ where } P^{\rho_i \hat{A}_i,\pi}_{kind,t_1} \\ \quad \textbf{pre } Pre^{\rho_i \hat{A}_i,\pi}_{kind} \\ \quad \textbf{eff } Prog^{\rho_i \hat{A}_i,\pi}_{kind} \textbf{ ensuring } ensuring^{\rho_i \hat{A}_i,\pi}_{kind} \end{array}\right]$$

...

Figure 14.2: General form of the expansion of the automaton for component C_i, obtained from the desugared definition $\hat{A}_i$ of the automaton on which C_i is based

tutions from the per-component substitutions. We also omit from these tables identity mappings. For example, we omit from Table 14.6 the identity mapping of `i1:Int` to itself due to rule 7 in $\sigma^{\texttt{P,overflow}}$. The resulting component automata are shown in Figures 14.3–14.5.

```
automaton C(nProcesses:Int, n:Int)
  signature
    input send(n1, n2:Int, m:Int) where n1 = n ∧ n2 = n+1
    output receive(n1, n2:Int, m:Int) where n1 = n ∧ n2 = n+1
  states contents:Set[Int] := {}
  transitions
    input send(n1, n2, m) where n1 = n ∧ n2 = n+1
      eff C[n].contents := insert(m, C[n].contents)
    output receive(n1, n2, m) where n1 = n ∧ n2 = n+1
      pre m ∈ C[n].contents
      eff C[n].contents := delete(m, C[n].contents)
```

Figure 14.3: Sample instantiated component automaton `C`, obtained by applying the substitutions in Table 14.5 to the resorted automaton `Channel` in Figure 14.1

SUBSTITUTION	DOMAIN	RANGE	RULE
$\sigma^{\texttt{C}}$	`Channel:States[Channel,Int,Int]`	`C[n]:Map[Int,States[Channel,Int,Int]]`	rule 3
	`i:Int`	`n:Int`	rule 1
	`j:Int`	`(n+1):Int`	rule 1
$\sigma^{\texttt{C,send}}$	No additional substitutions		
$\sigma^{\texttt{C,receive}}$	No additional substitutions		

Table 14.5: Substitutions used to derive sample component automaton `C`. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

```
automaton P(nProcesses:Int, n:Int)
  signature
    input receive(n1, n2, m:Int) where n1 = n-1 ∧ n2 = n
    output send(n1, n2, m:Int) where n1 = n ∧ n2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(n1, n2, m) where n1 = n-1 ∧ n2 = n
      eff if P[n].val = 0 then P[n].val := m
          elseif m < P[n].val then
            P[n].toSend := insert(P[n].val, P[n].toSend);
            P[n].val := m
          elseif P[n].val < m then
            P[n].toSend := insert(m, P[n].toSend)
          fi
    output send(n1, n2, m) where n1 = n ∧ n2 = n+1
      pre m ∈ P[n].toSend
      eff P[n].toSend := delete(m, P[n].toSend)
    output overflow(i1, s; local t:Set[Int]) where i1 = n
      pre s = P[n].toSend ∧ n < size(s) ∧ P[n].t ⊆ s
      eff P[n].toSend := P[n].t
```

Figure 14.4: Sample instantiated component automaton `P`, obtained by applying the substitutions in Table 14.6 to the automaton `P` in Figure 12.8

SUBSTITUTION	DOMAIN	RANGE	RULE
$\sigma^{\texttt{P}}$	P:States[P]	P[n]:Map[Int,States[P]]	rule 3
$\sigma^{\texttt{P,send}}$	i1:Int	n1:int	rule 7
	i2:Int	n2:int	rule 7
	x:Int	m:int	rule 7
$\sigma^{\texttt{P,receive}}$	i1:Int	n1:int	rule 7
	i2:Int	n2:int	rule 7
	x:Int	m:int	rule 7
$\sigma^{\texttt{P,overflow}}$	P:Locals[P,overflow]	P[n]:Map[Int,Locals[P,overflow]]	rule 9

Table 14.6: Substitutions used to derive sample component automaton P. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

```
automaton W(nProcesses:Int)
  signature
    input overflow(i1:Int, s:Set[Int]) where i1 ∈ between(1, nProcesses)
    output found(i1:Int) where i1 ∈ between(1, nProcesses)
  states seen:Array[Int,Bool] := constant(false)
  transitions
    input overflow(i1, s; local s2:Set[Int])
            where s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s)
      eff if s = W.s2 ∪ {i1} then W.seen[i1] := true
          elseif ¬(i1 ∈ s) then W.seen[i1] := false
          fi
    output found(i1)
      pre W.seen[i1]
```

Figure 14.5: Sample instantiated component automaton W, obtained by applying the substitutions in Table 14.7 to the resorted automaton Watch in Figure 14.1

SUBSTITUTION	DOMAIN	RANGE	RULE
$\sigma^{\texttt{W}}$	Watch:States[Watch,Int]	W:States[Watch,Int]	rule 2
	what:Set[Int]	between(1, nProcesses)	rule 1
$\sigma^{\texttt{W,overflow}}$	Watch:Locals[Watch,Int,overflow]	W:Locals[Watch,Int,overflow]	rule 8
	x:Int	i1:int	rule 7
$\sigma^{\texttt{W,found}}$	x:Int	i1:Int	rule 7

Table 14.7: Substitutions used to derive sample component automaton W. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

Syntactic Element	Original	Desugared	Resorted	Expanded	Component
Automaton	A_i	$\hat{A}_i$	$\rho_i \hat{A}_i$	$\sigma_i \rho_i \hat{A}_i$	C_i
General form	Figure 11.1	Figures 12.3, 12.5, 12.7		Figure 14.2	
Automaton parameters	$types^{A_i}$, $vars^{A_i}$		$types^{A_i}$, $\rho_i vars^{A_i}$	$types^{D}$, $vars^{D}$, $vars^{D,C_i}$	
Action parameters	$params_{kind}^{A_i,\pi}$	$vars^{\hat{A}_i,\pi}$	$\rho_i vars^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i vars^{\hat{A}_i,\pi}$	$vars^{D,\pi}$
Signature **where** predicates	$P_{kind}^{A_i,\pi}$	$P_{kind}^{\hat{A}_i,\pi}$	$\rho_i P_{kind}^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i P_{kind}^{\hat{A}_i,\pi}$	$P_{kind}^{C_i,\pi}$
State variables	$stateVars^{A_i}$		$\rho_i stateVars^{A_i}$		$stateVars^{C_i}$
Aggregate variable name	A_i			C_i	
Aggregate state sort	$States[A_i, types^{A_i}]$		$States[A_i, actualTypes^{D,C_i}]$		
Aggregate local sort	$Locals[A_i, types^{A_i}, kind, \pi, t_j]$	$Locals[A_i, types^{A_i}, \pi]$	$Locals[A_i, actualTypes^{D,C_i}, \pi]$		
Initial state variables values	$initVals^{A_i}$		$\rho_i initVals^{A_i}$	$\sigma_i \rho_i initVals^{A_i}$	$initVals^{C_i}$
initially predicate	$P_{init}^{A_i}$	$P_{init}^{\hat{A}_i}$	$\rho_i P_{init}^{\hat{A}_i}$	$\sigma_i \rho_i P_{init}^{\hat{A}_i}$	$P_{init}^{C_i}$
Transition parameters	$params_{kind,t_j}^{A_i,\pi}$	$vars^{\hat{A}_i,\pi}$	$\rho_i vars^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i vars^{\hat{A}_i,\pi}$	$vars^{D,\pi}$
local variables	$localVars_{kind,t_j}^{A_i,\pi}$	$localVars_{kind}^{\hat{A}_i,\pi}$	$\rho_i\ localVars_{kind}^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i\ localVars_{kind}^{\hat{A}_i,\pi}$	$localVars_{kind}^{C_i,\pi}$
Transition **where** predicates	$P_{kind,t_j}^{A_i,\pi}$	$P_{kind,t_1}^{\hat{A}_i,\pi}$	$\rho_i P_{kind,t_1}^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i P_{kind,t_1}^{\hat{A}_i,\pi}$	$P_{kind,t_1}^{C_i,\pi}$
Transition **pre** predicates	$Pre_{kind,t_j}^{A_i,\pi}$	$Pre_{kind}^{\hat{A}_i,\pi}$	$\rho_i Pre_{kind}^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i Pre_{kind}^{\hat{A}_i,\pi}$	$Pre_{kind}^{C_i,\pi}$
Transition **eff** programs	$Prog_{kind,t_j}^{A_i,\pi}$	$Prog_{kind}^{\hat{A}_i,\pi}$	$\rho_i Prog_{kind}^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i Prog_{kind}^{\hat{A}_i,\pi}$	$Prog_{kind}^{C_i,\pi}$
Transition **eff** predicates	$ensuring_{kind,t_j}^{A_i,\pi}$	$ensuring_{kind}^{\hat{A}_i,\pi}$	$\rho_i ensuring_{kind}^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i ensuring_{kind}^{\hat{A}_i,\pi}$	$ensuring_{kind}^{C_i,\pi}$
`Channel`	Figure 10.1	Figure 12.8	Figure 14.1	Figure 14.3	
`P`	Figure 10.2	Figure 12.8		Figure 14.4	
`Watch`	Figure 10.3	Figure 12.8	Figure 14.1	Figure 14.5	

Table 14.8: Stages in expanding components C_i of a composite automaton D.

Chapter 15

Expanding composite automata

Composition is, for the most part, an effort of slow diligence and steady perseverance, to which the mind is dragged by necessity or resolution, and from which the attention is every moment starting to more delightful amusements.

— Samuel Johnson [63]

In this chapter, we present the main contribution of Part II of this dissertation. We show how to expand a composite IOA program into an equivalent primitive IOA program. Section 15.1 reviews our assumptions about the form of the components of the composite automaton, and Section 15.2 describes a simplification of the structure of **hidden** statements, obtained by combining all clauses for a single action into a single clause.

In Section 15.3, we define the expansion of the signature of a composite automaton to primitive form. Section 15.4 gives first-order logic formulas for the semantic proof obligations we introduced in Section 13.4. These include compatibility requirements for component automata. In Section 15.5, we define the expansion of the **initially** predicate on the states of a composite automaton. In Sections 15.6–15.9, we define the expansion of the transitions of a composite automaton.

15.1 Expansion assumptions

We expand a composite automaton D into primitive form by combining elements of its components $C_1, \ldots, C_n$. We assume each component automaton A_i has been desugared to satisfy the restrictions in Section 12.5, resorted to produce an automaton $\rho_i \hat{A}_i$ as described in Section 13.2 and 14.1, and transformed as described in Section 14.4 to produce an automaton $\sigma_i \rho_i \hat{A}_i = C_i$. In particular, for

each component automaton C_i, we assume the following.

- No **const** parameters appear in the signature.
- Each appearance of an action π in the signature is parameterized by the canonical action parameters $vars^{D,\pi}$.
- Each transition definition of an action π is parameterized by the canonical action parameters $vars^{D,\pi}$.
- Each transition definition of an action π is further parameterized by the canonical sequence $\sigma_{i,\pi}\rho_i\ localVars^{\hat{A}_i,\pi}$ of local variables for that component.
- Each action has at most one transition definition of each kind.
- Every state, post-state, local variable, or post-local variable reference is of the unambiguous form $C_i.x$, $C_i'.x$, $C_i[vars^{D,C_i}].x$, or $C_i'[vars^{D,C_i}].x$.

15.2 Desugaring hidden statements of composite automata

The syntax for composite IOA programs as described in Chapter 13 provides programmers with flexibility of expression that can complicate expansion into primitive form. Hence, as with primitive automata, it is helpful to consider equivalent composite IOA programs that conform to a more limited "desugared" syntax. As discussed later in this section, **where** clauses of composite automaton **hidden** statements and of component transitions are combined during expansion. Thus, **hidden** statements must be desugared into a form analogous to that of a desugared transition. In particular, we desugar composite automata with **hidden** statements to have the following two properties.

- Each **hidden** clause for an action π is parameterized by the canonical action parameters $vars^{D,\pi}$.
- There is at most one **hidden** clause for each action π.

The static checks described in Section 13.3 ensure that the number and sorts of terms in $params_{hide_p}^{D,\pi_p}$ are the same as the number and sorts of variables in $vars^{D,\pi_p}$. If no variable in $vars^{D,\pi_p}$ occurs freely in $params_{hide_p}^{D,\pi_p}$ (*i.e.*, if $vars^{D,\pi_p}$ and $vars_{hide_p}^{D,\pi_p}$ are disjoint), then we can desugar the clause

$$\pi_p(params_{hide_p}^{D,\pi_p}) \textbf{ where } H_{hide_p}^{D,\pi_p}$$

by replacing $params_{hide_p}^{D,\pi_p}$ by $vars^{D,\pi_p}$, reintroducing $vars_{hide_p}^{D,\pi_p}$ as existentially quantified variables in the **where** clause, and adding conjuncts to the **where** clause to equate $vars^{D,\pi_p}$ with the old

parameters. This results in the desugaring

$$\pi_p(vars^{D,\pi_p}) \textbf{ where } \exists\ vars^{D,\pi_p}_{hide_p} \left(H^{D,\pi_p}_{hide_p} \ \wedge\ vars^{D,\pi_p} = params^{D,\pi_p}_{hide_p} \right).$$

Notice that introducing $vars^{D,\pi_p}_{hide_p}$ as existentially quantified variables is analogous to introducing $vars^{A,\pi}_{in,t_j}$ as local variables when desugaring transition parameters, as described in Section 12.1.

If $vars^{D,\pi_p}$ and $vars^{D,\pi_p}_{hide_p}$ are not disjoint, we define a substitution σ^{hide}_p that maps the intersection of these two sets to a set of fresh variables, and we desugar the **hidden** clause as

$$\pi_p(vars^{D,\pi_p}) \textbf{ where } \exists\ \sigma^{hide}_p vars^{D,\pi_p}_{hide_p} \left(\sigma^{hide}_p H^{D,\pi_p}_{hide_p} \ \wedge\ vars^{D,\pi_p} = \sigma^{hide}_p params^{D,\pi_p}_{hide_p} \right).$$

We simplify each existentially qualified **where** clause produced by the above transformations by dropping any existential quantifier, such as $\exists$ `i:Int` in the example, that introduces a variable equated to a term, as in `i = x` in the example, in the conjunction $vars^{D,\pi_p} = \sigma^{hide}_p params^{D,\pi_p}_{hide_p}$, and also by dropping the equating conjunct from that conjunction. We denote the resulting simplification of the **where** clause by $H^{D,\pi}_{hide_p,canon}$.

Following this clause-by-clause canonicalization, we combine all clauses in the **hidden** statement that apply to a single action π into one disjunction. This step is analogous to the combining step for transition definitions in Section 12.3. For example, if $\pi_p = \pi_q = \pi$, then the clauses

$$\pi_p(vars^{D,\pi_p}) \textbf{ where } H^{D,\pi_p}_{hide_p,canon}$$
$$\pi_q(vars^{D,\pi_q}) \textbf{ where } H^{D,\pi_q}_{hide_q,canon}$$

become the single clause

$$\pi(vars^{D,\pi}) \textbf{ where } H^{D,\pi_p}_{hide_p,canon} \ \vee\ H^{D,\pi_q}_{hide_q,canon}.$$

We denote this combined **where** clause by $H^{D,\pi}$.

15.3 Expanding the signature of composite automata

In the composite automaton D, actions that are internal to some component are internal actions of the composition, actions that are outputs of some component and are not hidden are output actions of the composition, and actions that are inputs to some components but outputs to none are input actions of the composition. The **where** clause predicates $P^{D,\pi}_{kind}$ express these facts in the signature of the expanded automaton $DExpanded$. We construct these predicates by defining subformulas, $P^{D,C_i,\pi}_{kind}$ and $Prov^{D,\pi}_{kind}$, which describe the actions components contribute to the composition. We

automaton $DExpanded(types^D, vars^D)$
  signature
    kind $\pi(vars^{D,\pi})$ **where** $P^{D,\pi}_{kind}$
    ...

Figure 15.1: General form of the signature in the expansion of a composite automaton

combine these formulas and the **where** predicate from any applicable **hidden** clause (*i.e.*, $H^{D,\pi}$), to account for the subsumption of input actions by output actions and for hiding output actions. The final result consists of the three predicates $P^{D,\pi}_{in}$, $P^{D,\pi}_{out}$, and $P^{D,\pi}_{int}$.

All free variables that appear in these predicates are among the composite automaton parameters $vars^D$ and the canonical action parameters $vars^{D,\pi}$. Figure 15.1 shows the general form of the expanded signature. Below, we explain how to construct these predicates. (See Section 16.2 for an example application of the process to composite automaton `Sys` defined in Example 10.4.)

15.3.1 Subformulas for actions contributed by a component

In order for an action **kind** $\pi(vars^{D,\pi})$ to be defined in D, it must be defined in some component. An action is defined in a component C_i of D if, given action parameters $vars^{D,\pi}$ there are component parameters $vars^{D,C_i}$ that satisfy both the component **where** clause P^{D,C_i} and the action **where** clause $P^{C_i,\pi}_{kind}$ for π in the signature of C_i. Hence we define

$$P^{D,C_i,\pi}_{kind} ::= \exists vars^{D,C_i}\ (P^{D,C_i} \wedge P^{C_i,\pi}_{kind}),$$

which is satisfied by $vars^{D,\pi}$ if and only if $\pi(vars^{D,\pi})$ is an action of type *kind* in component C_i of D.

It is important to note that the type of the action $\pi(vars^{D,\pi})$ in D may be different from the type of π in some, or even all, the components contributing the action to the composition. Output actions in one instance of one component may subsume inputs in another, and output actions may be hidden as internal actions in the composition. We say that *kind* is the *provisional kind* of $\pi(vars^{D,\pi})$ in D when an action of that kind is contributed to the composition by some component. Hence we define the predicate $Prov^{D,\pi}_{kind}$ as follows:

$$Prov^{D,\pi}_{kind} ::= \bigvee_{1 \le i \le n} P^{D,C_i,\pi}_{kind}.$$

15.3.2 Signature predicates

We account for subsumed inputs and hidden outputs in the signature of *DExpanded* by appending special case formulas to the predicates $Prov^{D,\pi}_{kind}$ to form the signature predicates $P^{D,\pi}_{kind}$. The three

cases we must consider are:

- An action $\pi(vars^{D,\pi})$ is an output action of D if and only if it is an output action in some component C_i of D and is not hidden in D.
- An action $\pi(vars^{D,\pi})$ is an input action of D if and only if it is an input action in some component C_i of D, but not an output action in any component of D.
- An action $\pi(vars^{D,\pi})$ is an internal action of D if and only if it is an internal action, or a hidden output action, in some component C_i of D.

Translating these requirements into first-order logic, we derive the following definitions for the signature predicates of $DExpanded$:

- $P_{out}^{D,\pi} ::= Prov_{out}^{D,\pi} \wedge \neg H^{D,\pi}$
- $P_{in}^{D,\pi} ::= Prov_{in}^{D,\pi} \wedge \neg Prov_{out}^{D,\pi}$
- $P_{int}^{D,\pi} ::= Prov_{int}^{D,\pi} \vee \left(Prov_{out}^{D,\pi} \wedge H^{D,\pi} \right)$.

15.4 Semantic proof obligations, revisited

We are now ready to formalize the following proof obligations on composite automata introduced in Section 13.4.

- ✓ Only output actions may be hidden.
- ✓ The components of a composite automaton have disjoint sets of output actions.
- ✓ The set of internal actions for any component is disjoint from the set of all actions of every other component.

Below we give corresponding formulas in first-order logic that must be verified for a composite IOA program to represent a valid I/O automaton. In order to express the latter two of these obligations in first-order logic, we break each of them into two parts. First, we consider different components from different clauses of the **components** statement (*i.e.*, $C_i \neq C_j$). Second, we consider instances of the same parameterized component distinguished only by parameter values (*i.e.*, $C_i[vars^{D,C_i}] \neq C_i[vars'^{D,C_i}]$). We use these formulas to help construct the expansion of transitions of composite automata in Sections 15.7–15.9.

15.4.1 Hidden actions

The first of these obligations is just the requirement that

- ✓ $H^{D,\pi} \Rightarrow Prov_{out}^{D,\pi}$.

15.4.2 Output actions

For output actions, we first require that different parameterized components have disjoint sets of output actions. Formally, we say that for all distinct components C_i and C_j of D, all values of the action parameters $vars^{D,\pi}$ for π, all values of the composite automaton parameters $vars^D$, and all values of the component parameters $vars^{D,C_i}$ and $vars^{D,C_j}$, we require that

$$\checkmark \quad \neg P_{out}^{D,C_i,\pi} \vee \neg P_{out}^{D,C_j,\pi} \tag{15.1}$$

Second, we require that different instances of the same parameterized component have disjoint sets of output actions. That is, for each component C_i of D, all values of the action parameters $vars^{D,\pi}$ for π, all values of the individual parameters $vars^D$ of the composite automaton, and all pairs of values of the component parameters $vars^{D,C_i}$ and $vars'^{D,C_i}$, we require that

$$\checkmark \quad \left(P^{D,C_i} \wedge P'^{D,C_i} \wedge P_{out}^{C_i,\pi} \wedge P'^{C_i,\pi}_{out}\right) \Rightarrow vars^{D,C_i} = vars'^{D,C_i} \tag{15.2}$$

where $P'^{C_i,\pi}_{out}$ is $P_{out}^{C_i,\pi}$ evaluated on $vars'^{D,C_i}$.

In Example 10.4, these requirements are satisfied trivially, because the output actions in the different components of `Sys` have different labels. However, the composition

```
automaton BadSys1
  components P1[n:Int] where 0 < n ∧ n < 10;
             P2: P(5)
```

would violate the first requirement, because components `P1[5]` and `P2` share an output action, and the composition

```
automaton BadSys2
  components W[what:Set[Int]]: Watch(Int, what)
             where what = between(1,1) ∨ what = between(1,2)
```

would violate the second requirement because components `W[[1]]` and `W[[1,2]]` both have `found(1)` as an output action.

15.4.3 Internal actions

Similarly, we break the last of these semantic proof obligations, which concerns internal actions, into two parts. We first require that internal actions are defined in one component only for parameter values where no action is defined in any other component. Formally, we say that for all distinct components C_i and C_j of D, all values of the action parameters $vars^{D,\pi}$, and all values of the composite automaton non-type parameters $vars^D$, we require that

$$\checkmark \quad P_{int}^{D,C_i,\pi} \Rightarrow \neg P_{all}^{D,C_j,\pi} \tag{15.3}$$

where $P_{all}^{D,C_j,\pi}$ is the disjunction of $P_{in}^{D,C_j,\pi}$, $P_{out}^{D,C_j,\pi}$, and $P_{int}^{D,C_j,\pi}$.

Second, we require that internal actions of one instance of a parameterized component are defined only for parameter values where no action is defined in any other instance of that component. That is, for each component C_i of D, all values of the action parameters $vars^{D,\pi}$, all values of the composite automaton non-type parameters $vars^{D}$, and all pairs of values of component non-type parameters $vars^{D,C_i}$ and $vars'^{D,C_i}$, we require that

$$\checkmark \quad \left(P^{D,C_i} \wedge P'^{D,C_i} \wedge P_{int}^{C_i,\pi} \wedge P'^{C_i,\pi}_{all}\right) \Rightarrow vars^{D,C_i} = vars'^{D,C_i}, \tag{15.4}$$

where $P'^{C_i,\pi}_{all}$ is the disjunction of $P'^{C_i,\pi}_{in}$, $P'^{C_i,\pi}_{out}$, and $P'^{C_i,\pi}_{int}$ and where the primed form of each predicate is the evaluation of the predicate on $vars'^{D,C_i}$.

Note, although allowed by obligation 15.4, the cases where $P_{int}^{C_i,\pi} \wedge P_{in}^{C_i,\pi}$ or $P_{int}^{C_i,\pi} \wedge P_{out}^{C_i,\pi}$ hold are already disallowed by semantic proof obligations 12.2 and 12.3, respectively.

Claim 15.1 (Signature compatibility) Semantic proof obligations 15.1–15.4 taken together with the signature **where** predicates $P_{kind}^{D,\pi}$ imply that *DExpanded* fulfills the semantic proof obligations for primitive automata 12.1–12.3.

In Sections 15.7–15.9, we argue that remaining obligations for primitive automata (12.4 and 12.5) are discharged by the transition **where** clauses of *DExpanded*.

15.5 Expanding initially predicates of composite automata

In Section 13.2, we described the state variables of a composite automaton D. Corresponding to each component C_i is a state variable C_i with sort $States[A_i, actualTypes^{D,C_i}]$ if C_i has no parameters and with sort `Map[`$types^{D,C_i}, States[A_i, actualTypes^{D,C_i}]$`]` otherwise. Here, we describe the construction of an **initially** predicate that constrains the initial values of these state variables. This predicate is a conjunction of clauses, one per unparameterized component and two per parameterized component.

If a component C_i is not parameterized (*i.e.*, the state variable C_i is a tuple, not a map), then a single clause asserts that, for all values of the component parameters for which the component is defined (*i.e.*, when P^{D,C_i} is true), each element of the tuple has an appropriate initial value. Furthermore, the clause asserts that, when P^{D,C_i} is true, the tuple as a whole satisfies the **initially** predicate $P_{init}^{C_i}$ of the component. In order to account for initial values specified as nondeterministic choices, we proceed as follows. Let

- X_i be the set of indices k of state variable declarations of the form

  $$x_k{:}T_k \ := \ \textbf{choose } v_k{:}T_k \textbf{ where } P_{init,k}^{C_i}$$

  in the definition of the component C_i,

- $cVars^{C_i}$ be a set of distinct fresh variables $v'_k{:}T_k$, one for each k in X_i,
- $*initVals^{C_i}$ be $initVals^{C_i}$ with each of the above **choose** expressions replaced by the corresponding $v'_k{:}T_k$ for each k in X_i, and
- $*P^{C_i}_{init,k}$ be $P^{C_i}_{init,k}$ with v'_k substituted for v_k when $k \in X_i$ and the predicate *true* otherwise.

Then we formulate the clause (shown in Figure 15.2) corresponding to C_i in the **initially** predicate of *DExpanded* by factoring out, and existentially qualifying, the variables (*i.e.*, $cVars^{C_i}$) used to choose nondeterministic values for the state variables of the component automaton C_i.

When a component C_i is parameterized (*i.e.*, the state variable C_i is a map, not a tuple), then there are two clauses for the component. The first is analogous to the single clause for the simple case in which the state variable is a tuple, but now it asserts that each element of each tuple in the map has an appropriate initial value and that, when P^{D,C_i} is true, the map as a whole satisfies the **initially** predicate of the component. The second clause asserts that the map is defined exactly for the values of the component parameters for which the component itself is defined (*i.e.*, when P^{D,C_i} is true). This second clause is also asserted automatically as an invariant of the automaton. That is, no transition either extends or reduces the domain over which the map is defined. Figure 15.2 summarizes these two cases and the invariant.

15.6 Combining local variables of composite automata

Just as it helped to collect the local variables from all transition definitions for an action π when desugaring a primitive automaton (see Section 12.3), it helps to collect the local variables from the transitions definitions from different components for an action π when expanding the definition of a composite automaton. Hence, we parameterize every transition definition by n per-component aggregate local variables that are named for the components $C_1, \ldots, C_n$ just as the n per-component aggregate state variables are named for those components (see Section 13.2).

The sort of each per-component local variable depends on the name of the action and the parameterization of the component. If the component C_i has no parameters, then the aggregate local variable C_i has sort $Locals[A_i, actualTypes^{D,C_i}, \pi]$. On the other hand, if the component C_i has parameters, then the aggregate local variable C_i has sort `Map[`$types^{D,C_i}$, $Locals[A_i, actualTypes^{D,C_i}, \pi]$`]`, where $types^{D,C_i}$ is the sequence of types of the variables in $vars^{D,C_i}$.

We define $localVars^{D,\pi}$ to be the sequence of the per-component local variables $C_1, \ldots, C_n$. If a transition π has no local variables in component C_i or if π is not a transition in component C_i, we omit C_i from $localVars^{D,\pi}$. We also define the sort $Locals[D, types^D, \pi]$ to be a tuple sort with selection operators that are named, typed, and have values in accordance with the variables in $localVars^{D,\pi}$.

states

$\ldots,$

C_i: $States[A_i, actualTypes^{D,C_i}]$, % if $vars^{D,C_i}$ is empty

$\ldots,$

C_j: $\texttt{Map}[vars^{D,C_j}, States[A_j, actualTypes^{D,C_j}]]$, % if $vars^{D,C_j}$ is not empty

$\ldots$

initially

$\ldots \wedge$

$P^{D,C_i} \Rightarrow \exists cVars^{C_i} \left(P^{C_i}_{init} \wedge C_i.stateVars^{C_i} = *initVals^{C_i} \wedge \bigwedge_{k \in X_i} *P^{C_i}_{init,k} \right) \wedge$

$\ldots \wedge$

$\forall vars^{D,C_j} \left(P^{D,C_j} \Rightarrow \exists cVars^{C_j} (P^{C_j}_{init} \wedge C_j[vars^{D,C_j}].stateVars^{C_j} = *initVals^{C_j} \wedge \bigwedge_{k \in X_j} *P^{C_j}_{init,k}) \right) \wedge$

$\forall vars^{D,C_j} \left(P^{D,C_j} \Leftrightarrow \texttt{defined}(C_j[vars^{D,C_j}]) \right) \wedge$

$\ldots$

invariant of $DExpanded$:

$\ldots;$

$\forall vars^{D,C_j} \left(P^{D,C_j} \Leftrightarrow \texttt{defined}(C_j[vars^{D,C_j}]) \right);$

$\ldots$

Figure 15.2: General form of the states in the expansion of a composite automaton

15.7 Expanding input transitions

Composition combines the transitions for identical input actions in different component automata into a single atomic transition. An input transition is defined for an action π exactly for those values of $vars^{D,\pi}$ that satisfy the signature **where** predicate $P_{in}^{D,\pi}$. Figure 15.3 shows the general form for the definition of a combined input transition based on this observation. Below, we discuss the definitions of the **where**, **eff**, and **ensuring** clauses which appear in that figure.

Each of these clauses also appears as part of the expanded transitions for output and internal transitions, so we name them $P_{in,t_1}^{D,\pi}$, $Prog_{in}^{D,\pi}$, and $ensuring_{in}^{D,\pi}$, respectively, and include them in the figures for the output and internal transitions only by reference. In those transitions, $P_{in,t_1}^{D,\pi}$ refers only to the predicate explicitly appearing in Figure 15.3. That is, without the implicitly conjoined signature predicate $P_{in}^{D,\pi}$.

transitions
...
input $\pi(vars^{D,\pi}$; **local** $localVars^{D,\pi})$ **where** $\bigwedge_{1\leq i\leq n} P_{in,t_1}^{D,C_i,\pi}$
eff
...
% When $vars^{D,C_i}$ is empty
if $P^{D,C_i} \wedge P_{in}^{C_i,\pi}$ **then** $Prog_{in}^{C_i,\pi}$ **fi**;
...
% When $vars^{D,C_j}$ is not empty
for $vars^{D,C_j}$ **where** $P^{D,C_j} \wedge P_{in}^{C_j,\pi}$ **do**
$Prog_{in}^{C_j,\pi}$
od;
...
ensuring $\bigwedge_{1\leq i\leq n} ensuring_{in}^{D,C_i,\pi}$

Figure 15.3: General form of an input transition in the expansion of a composite automaton

15.7.1 where clause

Since there is only one input transition for the action π in *DExpanded*, the expanded transition **where** clause trivially satisfies semantic proof obligation 12.5 and its only functional role is to define the initial values of the local variables $localVars^{D,\pi}$ that correspond to a given sequence of action parameters $vars^{D,\pi}$. While the signature **where** predicate $P_{in}^{D,\pi}$ need only establish that there exists *some* instance of *some* component that contributes an input action $\pi(vars^{D,\pi})$, the transition **where** predicate must define local variable initial values for *each* contributing instance of *all* contributing components.

We define the input transition **where** clause $P^{D,\pi}_{in,t_I}$ by constructing subformulas $P^{D,C_i,\pi}_{in,t_I}$. Each such subformula constrains the initial value of one local variable C_i of contributing component C_i. The **where** clause shown in Figure 15.3 is then just the conjunction of these predicates $P^{D,C_i,\pi}_{in,t_I}$ for all components.

The subformula $P^{D,C_i,\pi}_{in,t_I}$ is the implication that for each instance of the component that contributes to the transition, the local variable C_i satisfies the proper initial constraints. The initial value of local variable C_i in $localVars^{D,\pi}$ is properly constrained when it satisfies the **where** clause $P^{C_i,\pi}_{in,t_I}$ for the input transition definition of π in component C_i (for the given values of the component parameters $vars^{D,C_i}$ and action parameters $vars^{D,\pi}$). Thus, the consequent of the subformula implication is $P^{C_i,\pi}_{in,t_I}$.

When the component is parameterized, the local variable C_i is a map and each entry $C_i[vars^{D,C_i}]$ in that map corresponds to the aggregate local variable for one instance of the component. In this case, the initial values for entries corresponding to all contributing instances must initialized. An instance of component C_i contributes to the transition $\pi(vars^{D,\pi})$ when component parameters $vars^{D,C_i}$ satisfy both the component **where** clause P^{D,C_i} and the signature **where** clause $P^{C_i,\pi}_{in}$ in that component (for the given values of the action parameters $vars^{D,\pi}$). Thus, the antecedent of the implication is the conjunction of these two predicates. To cover all instances, the implication is universally quantified over all values of the component parameters $vars^{D,C_i}$. Hence, we define

$$P^{D,C_i,\pi}_{in,t_I} ::= \forall\, vars^{D,C_i} \left(\left(P^{D,C_i} \wedge P^{C_i,\pi}_{in} \right) \Rightarrow P^{C_i,\pi}_{in,t_I} \right).$$

Since component C_i satisfies the semantic proof obligation 12.4, there must exists a value for local variable C_i that satisfies the above consequent whenever the antecedent holds. Thus, the implication is always true when read with the existential quantifier over the local variables $localVars^{D,\pi}$ that is implicit in the transition header. Thus, $DExpanded$ also (trivially) satisfies semantic proof obligation 12.4 for input transitions, since whenever the input action $\pi(vars^{D,\pi})$ is defined in the signature of $DExpanded$, the input transition $\pi(vars^{D,\pi})$ is also defined.

Notice that for each distinct value of $vars^{D,C_i}$ the predicate $P^{C_i,\pi}_{in,t_I}$ mentions a distinct local variable C_i or $C_i[vars^{D,C_i}]$ in $localVars^{D,\pi}$. So, the truth values of instantiations of the the implication are independent even though there is only one existential instantiation of the local variables $localVars^{D,\pi}$.

However, the fact that the implication is always true does not mean that it is equivalent to omit the expanded transition **where** clause. It is a consequence of the expanded signature **where** clause $P^{D,\pi}_{in}$ that some value of $vars^{D,C_i}$ satisfies the above implication antecedent. In that case the **where** clause asserts that the initial value of the relevant local variable must satisfy the contributing component transition **where** predicate $P^{C_i,\pi}_{in,t_I}$.

When the component is not parameterized, $P_{in,t_1}^{D,C_i,\pi}$ reduces to $P_{in,t_1}^{C_i,\pi}$. To see this, first, note that the universal quantifier simplifies away for lack of variables to quantify. Second, note that P^{D,C_i} and $P_{in}^{C_i,\pi}$ are true whenever $P_{in}^{D,\pi}$ is true. So the implication reduces to just the consequent.

Since the only functional role of the **where** clause is to define the initial values of the local variables $localVars^{D,\pi}$, when there are no local variables or when no local variable appears in any $P_{in}^{C_i,\pi}$, the **where** clause can be omitted altogether.

15.7.2 eff clause

The **eff** clause performs the effects of all input transitions of each contributing instance of all contributing components. It contains a conditional statement for each unparameterized component C_i of D and a loop statement for each parameterized component C_i of D.

The predicate in the conditional statement for an unparameterized component C_i (when implicitly conjoined with the **where** clause for the entire transition and **where** clause for the action in the automaton signature) is true if C_i contributes an input transition for π to the composite automaton D. In that case, the body of the conditional statement executes the program in the **eff** clause in the transition definition for π in C_i.

The situation is slightly more complicated when the component C_i is parameterized, because the transition must execute the effects of all instances of the component that contribute to the action. Thus, the **eff** clause loops over all the different values of the component parameters $vars^{D,C_i}$ that satisfy the component **where** clause P^{D,C_i} and the signature **where** clause $P_{in}^{C_i,\pi}$ in that component to execute the program in the **eff** clause in the transition for π in that instance of component C_i. Notice that each instance of a contributing component C_i (corresponding to one iteration of the loop for C_i) manipulates a distinct tuple of local variables $C_i[vars^{D,C_i}]$.[1]

If only one unparameterized component C_i contributes to the input transition definition, the conditional statement for that component may be replaced by the **eff** clause in the transition definition for π in C_i itself because the guard is implied by $P_{in}^{D,\pi}$.

15.7.3 ensuring clause

The **ensuring** predicate must be true if and only if the **ensuring** predicate from each contributing instance of all contributing components is true. That is, given the parameters $vars^{D,\pi}$, for each

[1] Currently, IOA syntax permits only a single single loop variable in **for** statements. However, if V is a sequence of variables $v_1, v_2, v_3, \ldots$, then it is simple to rewrite multi-variable loops such as the ones used in Figure 15.3

for V **where** p **do** g **od**

as nested single-variable loops using the inductive step

for v_1 **where** $\exists V' p$ **do**
  for V' **where** p **do** g **od**
od

where is the variable sequence $V' = v_2, v_3 \ldots$, p is a predicate and g is a program.

sequence of values of component parameters $vars^{D,C_i}$ of each component C_i that satisfies both the component **where** clause P^{D,C_i} and the signature **where** clause $P_{in}^{C_i,\pi}$ in that component, the value of the local variable C_i in $localVars^{D,\pi}$ must also satisfy the **ensuring** clause $ensuring_{in}^{C_i,\pi}$ for the input transition definition of π in C_i. Thus, we define the predicate $ensuring_{in}^{D,C_i,\pi}$ analogously to the the predicate $P_{in,t_1}^{D,C_i,\pi}$:

$$ensuring_{in}^{D,C_i,\pi} ::= \forall\, vars^{D,C_i} \left(\left(P^{D,C_i} \wedge P_{in}^{C_i,\pi} \right) \Rightarrow ensuring_{in,t_1}^{C_i,\pi} \right).$$

15.8 Expanding output transitions

We build up to the general form of expanded output transitions by first considering three specialized cases. The simplest case we consider is an output transition that appears in exactly one unparameterized component and in no component as an input transition. Second, we consider the expansion of an output transition when that sole contributing component is parameterized. Third, we extend our definitions to apply output transitions contributed by multiple components. Finally, the fully general expansion of output transitions covers the case where output actions and input actions share a name.

15.8.1 Output-only transition contributed by a single unparameterized component

We begin by considering the simplest case of an output transition $\pi(vars^{D,\pi})$ that appears in exactly one unparameterized component C_i and in no component as an input transition. That is, there is no component C_j, whose signature contains an input action $\pi(vars^{D,\pi})$. In this case, the expanded output transition does not need to be performed atomically with any input transition.

As there is only one transition contributing to the expansion, there is only one transition for the action $\pi(vars^{D,\pi})$ in *DExpanded*. Thus, the expanded transition **where** clause trivially satisfies semantic proof obligation 12.5 and its only functional role is to define the initial values of the local variable C_i that corresponds to a given sequence of parameters $vars^{D,\pi}$. In this case, simply reusing the component transition **where** clause $P_{out,t_1}^{C_i,\pi}$ as the expanded transition **where** clause gives the correct definition. In fact, the only difference between the expanded transition and the component transition in this simplest case is the way locals variables are declared in transition header. The aggregate local variable of the component transition becomes the sole local variable of the expanded transition. The resulting form is show in Figure 15.4.

transitions

...

output π(**local** $vars^{D,C_i}$, C_i:$Locals[C_i, actualTypes^{D,C_i}, \pi]$)

where $P^{C_i,\pi}_{out,t_1}$

pre $Pre^{C_i,\pi}_{out}$

eff $Prog^{C_i,\pi}_{out}$

ensuring $ensuring^{C_i,\pi}_{out}$

Figure 15.4: Expanded transition for an output action with no matching input actions, derived uniquely from a component C_i with no parameters.

15.8.2 Output-only transition contributed by a single parameterized component

When the component C_i has parameters the expansion is slightly more complicated. As in the previous case, no like-named input transitions exist in any component and, therefore, the expanded output transition does not need to be performed atomically with any input transition. Also like the previous case, there is only one transition definition for $\pi(vars^{D,\pi})$ in the expanded automaton, so the transition **where** clause trivially satisfies semantic proof obligation 12.5 and its only functional role is to define the initial values of the local variables. Unlike the previous case, the state and local variables C_i are maps rather than simple tuples and the contributing component parameters $vars^{D,C_i}$ are introduced as local variables.

The initial values of $vars^{D,C_i}$ need to be the correct indices for the relevant entry in the state and local variable maps. That is, $C_i[vars^{D,C_i}]$ should evaluate to the tuple derived from the aggregate variable of the contributing instance of the component. Note, the semantic proof obligation 15.2 requires that at most one instance of a component may contribute an output action $\pi(vars^{D,\pi})$. In fact, proof obligation 15.2 provides the formula for selecting the correct indices. The component parameters of the sole contributing instance uniquely satisfy both the component **where** clause P^{D,C_i} and the signature **where** clause $P^{C_i,\pi}_{out}$. Thus, these two predicates appear as conjuncts in the **where** clause.

Since at most one instance of component C_i contributes to the expanded transition, at most one entry in each of state and local variable maps C_i, corresponding to the aggregate variable of the contributing instance of the component, has any relevance to the transition. The other entries are completely ignored.[2] The initial values for that entry $C_i[vars^{D,C_i}]$ are those that satisfy the component transition **where** clause $P^{C_i,\pi}_{out,t_1}$. Thus, this predicate forms the last conjunct in the expanded **where** clause.

[2]In this special case, the references to local variable maps (rather than simple tuples) introduced by substitution $\sigma_{i,\pi}$ rule 9 in Section 14.3 are actually an unnecessary complication. However, they are required in the more general cases discussed below.

transitions

...

output $\pi(vars^{D,\pi}$; **local** $vars^{D,C_i}$, C_i:`Map`$[types^{D,C_i}, Locals[C_i, actualTypes^{D,C_i}, \pi]])$

where $P^{D,C_i} \wedge P_{out}^{C_i,\pi} \wedge P_{out,t_1}^{C_i,\pi}$

pre $Pre_{out}^{C_i,\pi}$

eff $Prog_{out}^{C_i,\pi}$

ensuring $ensuring_{out}^{C_i,\pi}$

Figure 15.5: Expanded transition for an output action with no matching input actions, derived uniquely from a parameterized component C_i.

The fact that at most one instance of component C_i contributes to the expanded transition also means the expanded definition for the transition of an output action π need not use a **for** statement, as does the expanded definition for the transition of an input action. Instead, the expanded definition simply reuses the **eff** clause of the sole contributing component transition. Similarly, the **pre** and **ensuring** clauses of the expanded transition are the same as those of the sole contributing component transition, as shown in Figure 15.5.

15.8.3 Output-only transitions contributed by multiple components

When an output action name appears in several components, it would be valid for the expanded composite automaton to include a separate output transition derived from each contributing component transition using the above definitions. Unfortunately, as we see below, this approach yields a code-size explosion multiplicative in the number of like-named input and output transitions. To avoid this code explosion, we define the expanded composite automaton to combine all like-named output transitions into a single output transition, as shown in Figure 15.6. An additional advantage of combining all like-named output transitions is that, once again, the expanded transition **where** clause trivially satisfies semantic proof obligation 12.5 and its only functional role is to define the initial values of the local variables.

In the expansion, we declare as local variables the parameters of each (contributing) component and the local variable C_i from each (contributing) component. As in the previous case, the semantic proof obligations for output actions given in Section 15.4 provide the key to defining the **where** clause. Obligation 15.1 requires that for any value of parameters $vars^{D,\pi}$, at most one disjunct of

$$\bigvee_{1 \le i \le n} P_{out}^{D,C_i,\pi} = \bigvee_{1 \le i \le n} \exists vars^{D,C_i} \, (P^{D,C_i} \wedge P_{out}^{C_i,\pi})$$

can be true. That is, at most one component may contribute an output transition $\pi(vars^{D,\pi})$. Since, all the component parameters $vars^{D,C_i}$ appear as local variables in the expanded transition header,

transitions

...

output $\pi(vars^{D,\pi}$; **local** $vars^{D,C_1}, \ldots, vars^{D,C_n},\ localVars^{D,\pi})$

where $\bigvee_{1\leq i\leq n}\left(P^{D,C_i}\ \wedge\ P_{out}^{C_i,\pi}\ \wedge\ P_{out,t_1}^{C_i,\pi}\right)$

pre $\bigvee_{1\leq i\leq n}\left(P^{D,C_i}\ \wedge\ P_{out}^{C_i,\pi}\ \wedge\ Pre_{out}^{C_i,\pi}\right)$

eff

if ...

elseif $P^{D,C_i}\ \wedge\ P_{out}^{C_i,\pi}$ **then** $Prog_{out}^{C_i,\pi}$

elseif ...

fi

ensuring $\bigwedge_{1\leq i\leq n}\left(P^{D,C_i}\ \wedge\ P_{out,t_1}^{C_i,\pi}\ \Rightarrow\ ensuring_{out}^{A,\pi}\right)$

Figure 15.6: Expanded transition for an output action with no matching input actions, contributed by several components

these variables are implicitly existentially quantified in the **where** clause. Therefore, in the expanded transition, the above obligation can be expressed simply as

$$\bigvee_{1\leq i\leq n}(P^{D,C_i}\ \wedge\ P_{out}^{C_i,\pi}).$$

Similarly, obligation 15.2 requires that at most one set of values for the component parameters $vars^{D,C_i}$ of that contributing component C_i satisfies the conjunction

$$P^{D,C_i}\ \wedge\ P_{out}^{C_i,\pi}.$$

That is, at most one instance of that component may contribute an output transition $\pi(vars^{D,\pi})$. Notice that this conjunction appears exactly in the previous obligation. In fact, we use the conjunction of the component **where** clause P^{D,C_i} of the contributing component and the signature **where** clause $P_{out}^{C_i,\pi}$ as a "guarding conjunction" for selecting the contributing instance of the contributing component throughout the expanded output transition.

In the **where** clause the guarding conjunction is paired with the corresponding component transition **where** clause and that triple conjunct is disjoined over all the components. Doing so has the effect that the initial values of the relevant local variable C_i (or its relevant map entry $C_i[vars^{D,C_i}]$) satisfies the component transition **where** clause whenever C_i is the contributing component.

Notice, it is a consequence of the expanded signature **where** clause $P_{out}^{D,\pi}$ that some value of $vars^{D,C_i}$ satisfies the guarding conjunction. Furthermore, since component C_i satisfies the semantic proof obligation 12.4, there must exists a value for local variable C_i that satisfies the consequent whenever the guarding conjunction is true. Therefore, whenever the output action $\pi(vars^{D,\pi})$ is

defined in the signature of *DExpanded*, the output transition $\pi(vars^{D,\pi})$ is also defined. Thus, *DExpanded* also satisfies semantic proof obligation 12.4 for output transitions.

In the precondition, the guarding conjunction is paired with the corresponding component precondition and that triple conjunct is disjoined over all the components. Thus, the expanded transition is enabled when there is a component for which all three of the transition precondition, the transition **where** clause, and the component **where** clause are true for the given parameters and initial local variable values. Checking the conjunction of all three predicates avoids enabling the transition when the **where** clause is satisfied by the transition from one component while the **pre** clause is satisfied by the transition of another component.

In the **eff** clause, the guarding conjunction selects the conditional branch containing the effects of the single contributing output transition that is defined for the given parameters. Similarly, the **ensuring** clause of the contributing output transition must be satisfied.

15.8.4 Output transitions subsuming input transitions (general case)

When both input and output transitions are defined and (the output transition is) enabled, the output transition subsumes the input transitions. That is, the input actions execute atomically with the output action. Just as we cannot statically decide that two input actions will never be simultaneously executed, we cannot, in general, statically decide that an input transition can never be subsumed by a like-named output transition. Therefore, each expanded output transition must include the effects of *all* like-named input transitions (appropriately guarded). (It is this fact that would cause the code-size explosion mentioned in the previous section were we to include a separate output transition derived from each contributing component transition.) Figure 15.7 shows the general form for expanding output transitions of composite automata.

In the cases where the output transition subsumes one or more input transition, the local variables from the instance(s) of the component(s) contributing the input transition(s) must be initialized by the expanded transition **where** clause. On the other hand, the **where** clause must still always be satisfiable when an output action is defined. As we argue in Section 15.7, the expanded input transition **where** predicate $P^{D,\pi}_{in,t_I}$ does exactly these two things. First, it requires the local variables derived from contributing input transitions to satisfy the **where** clauses of those transitions. Second, $P^{D,\pi}_{in,t_I}$ is satisfiable by some choice of values for $localVars^{D,\pi}$. Thus, we simply conjoin $P^{D,\pi}_{in,t_I}$ to the **where** clause developed in the previous case.

The **eff** clause selects the effects of the single contributing output transition that is defined for the given parameters and then performs all the effects of the subsumed input transitions by executing $Prog^{D,\pi}_{in}$. Each effect in $Prog^{D,\pi}_{in}$ is already guarded so as to occur only when the source transition contributes. Therefore, we simply append $Prog^{D,\pi}_{in}$ to the **eff** clause from the previous case. Similarly, the **ensuring** clause $ensuring^{D,\pi}_{in}$ can also be simply conjoined with the the **ensuring**

transitions

...

output $\pi(vars^{D,\pi}$; **local** $vars^{D,C_1}, \ldots, vars^{D,C_n},\ localVars^{D,\pi})$

where $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{out}^{C_i,\pi} \ \wedge \ P_{out,t_1}^{C_i,\pi} \right) \ \wedge \ P_{in,t_1}^{D,\pi}$

pre $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{out}^{C_i,\pi} \ \wedge \ Pre_{out}^{C_i,\pi} \right)$

eff

if ...

elseif $P^{D,C_i} \ \wedge \ P_{out}^{C_i,\pi}$ **then** $Prog_{out}^{C_i,\pi}$

elseif ...

fi;

$Prog_{in}^{D,\pi}$

ensuring $\bigwedge_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{out}^{C_i,\pi} \ \Rightarrow \ ensuring_{out}^{A,\pi} \right) \ \wedge \ ensuring_{in}^{D,\pi}$

Figure 15.7: General form of an output transition in the expansion of a composite automaton

clause from the previous case.

Note that, $Prog_{in}^{D,\pi}$ may, in fact, amount to a no-op in all executions. However, in general, this cannot be statically decided. Also note that the order of execution of the subsumed input transitions with respect to each other or to the enabled output transition does not matter. The semantic checks require that each conditional branch or **for** body executed in either the subsumed input transition or the remainder of the clause must be derived from distinct automata. These effects can alter only the value of state, local, or **choose** variables derived from the automaton contributing that effect. Furthermore, the effects can depend only on those same set of state, local, and **choose** variables or on the parameters of the transition. No effect can change a parameter value.

We define $Prog_{out}^{D,\pi}$ to be the program in the **eff** clause that combines the effects of output transitions and subsumed input transitions. Similarly, we define $ensuring_{out}^{D,\pi}$ to be the predicate that appears in the **ensuring** clause.

15.9 Expanding internal transitions

The basic form of expanded internal transitions is analogous to that of output actions. The most significant difference is that the internal transition expansion must account for output actions that are (potentially) hidden. So before we consider the general expansion for internal transitions, we build on the discussion of the expansion of output transitions above to consider the the simpler case of expanding transitions for internal actions when there are no **hidden** clauses for those actions. We then discuss how to generalize this construction to account for hidden output transitions.

transitions

...

internal $\pi(vars^{D,\pi}$; **local** $vars^{D,C_1}, \ldots, vars^{D,C_n},\ localVars^{D,\pi})$

where $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{int}^{C_i,\pi} \ \wedge \ P_{int,t_1}^{C_i,\pi} \right)$

pre

$\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{int}^{C_i,\pi} \ \wedge \ Pre_{int}^{C_i,\pi} \right)$

eff

if ...

elseif $P^{D,C_i} \ \wedge \ P_{int}^{C_i,\pi} \ \wedge$ **then** $Prog_{int}^{C_i,\pi}$

elseif ...

fi

ensuring $\bigwedge_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{int,t_1}^{C_i,\pi} \ \Rightarrow \ ensuring_{int}^{A,\pi} \right)$

Figure 15.8: Expanded transition for an internal action with no matching **hidden** clause

15.9.1 Internal-only transitions

The expanded form of the transition for an internal action when there is no **hidden** clause for that action follows a pattern similar to that of output transitions when there are no like-named input transitions. In that expansion, shown in Figure 15.8, we introduce local variables for the parameters of each contributing automaton as well as all the local variables from all the contributing transitions. Following reasoning analogous to the output case, we use the conjunction of the component **where** clause P^{D,C_i} of the contributing component and the signature **where** clause $P_{int}^{C_i,\pi}$ as the guarding conjunction for selecting the contributing instance of the contributing component throughout the expanded internal transition.

In the **where** clause, the guarding conjunction is paired with the component **where** clause for the contributing transition $P_{int,t_1}^{C_i,\pi}$ to initialize the local variable values. In the precondition, the guarding conjunction is paired with the **pre** predicate of the contributing transition. In the **eff** clause, the guarding conjunction selects the conditional branch containing the effects of the single contributing transition that is defined for the given parameters. In, the **ensuring** clause, the contributing transition **ensuring** clause must be satisfied when the guarding conjunction holds.

15.9.2 Internal transitions with hiding (general case)

The most important difference between the expansion for internal transitions and that for output transitions is that the internal transition expansions must account for output actions that are (potentially) hidden. We cannot, in general, statically decide whether the **hidden** predicate $H^{D,\pi}$ covers the output signature predicate $P_{out}^{D,\pi}$. Nor can we, in general, statically decide whether $H^{D,\pi}$ covers the **where** clause for any contributing transition $P_{out,t_1}^{C_i,\pi}$. Thus, each transition for each ac-

tion $\pi(vars^{D,\pi})$ mentioned by a **hidden** clause must be incorporated into the expanded composite automaton *twice*, once in an output transition and once in an internal transition.

One way to do this, would be to include two internal transitions for each transition $\pi(vars^{D,\pi})$. The first transition would be derived as in the previous section, ignoring any hidden output actions. The second transition would be a second copy of the expanded output transition $\pi(vars^{D,\pi})$. This transition would be identical to the general case output transition expansion except it would be labeled internal.

An alternative expansion is shown in Figure 15.9. This expansion follows the pattern of including just one transition of each kind. An advantage of having just one transition is that the expanded transition **where** clause trivially satisfies semantic proof obligation 12.5 and its only functional role is to define the initial values of the local variables.

Proof obligations 15.3 and 15.4 imply that, over all components, at most one of the conjunctions $P^{D,C_i} \wedge P^{C_i,\pi}_{int}$ and $P^{D,C_i} \wedge P^{C_i,\pi}_{out}$ can be true. So these conjunctions are used as the guarding conjunctions for the expanded transition. The former guards elements derived from internal component transitions. The latter guards elements derived from output component transitions.

In the **where** clause, each guarding conjunction is paired with the component **where** clause for the contributing transition $P^{C_i,\pi}_{kind,t_l}$ of matching kind to initialize the local variable values. Since a hidden output transition might also subsume a like-named input action, the **where** predicate also asserts $P^{D,\pi}_{in}$.[3] In the precondition, the guarding conjunction selects the appropriate component transition precondition $Pre^{C_i,\pi}_{int}$ or $Pre^{C_i,\pi}_{out}$ to satisfy. These latter disjuncts are abbreviated by referencing the expanded output **pre** predicate $Pre^{D,\pi}_{out}$. The **eff** clause selects the effects of the single contributing internal or output transition that is defined for the given parameters and then performs all the effects of the subsumed input transitions. The conditional selecting the effects of an internal action is shown in the figure. Effects derived from hidden output and hidden subsumed inputs are executed in the appended program $Prog^{D,\pi}_{out}$. Similarly, the **ensuring** clause from the previous case can be simply conjoined with expanded output transition **ensuring** clause $ensuring^{D,\pi}_{out}$

Notice, it is a consequence of the expanded signature **where** clause $P^{D,\pi}_{int}$ that some value of $vars^{D,C_i}$ satisfies one of the guarding conjunctions. Furthermore, since component C_i satisfies the semantic proof obligation 12.4, there must exists a value for local variable C_i that satisfies the consequent whenever a guarding conjunction is true. Therefore, whenever the internal action $\pi(vars^{D,\pi})$ is defined in the signature of $DExpanded$, the internal transition $\pi(vars^{D,\pi})$ is also defined. Thus, $DExpanded$ also satisfies semantic proof obligation 12.4 for internal transitions.

[3]We cannot simply conjoin $P^{D,\pi}_{out}$ to the transition **where** clause because $P^{D,\pi}_{in}$ would not distribute correctly.

transitions

...

internal $\pi(vars^{D,\pi};$ **local** $vars^{D,C_1}, \ldots, vars^{D,C_n},\ localVars^{D,\pi})$

where $\bigvee_{1 \le i \le n} \left(\left(P^{D,C_i} \ \wedge \ P_{int}^{C_i,\pi} \ \wedge \ P_{int,t_1}^{C_i,\pi} \right) \ \vee \ \left(P^{D,C_i} \ \wedge \ P_{out}^{C_i,\pi} \ \wedge \ P_{out,t_1}^{C_i,\pi} \right) \right) \ \wedge \ P_{in,t_1}^{D,\pi}$

pre

$\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{int}^{C_i,\pi} \ \wedge \ Pre_{int}^{C_i,\pi} \right) \ \vee \ Pre_{out}^{D,\pi}$

eff

if ...

elseif $P^{D,C_i} \ \wedge \ P_{int}^{C_i,\pi} \ \wedge$ **then** $Prog_{int}^{C_i,\pi}$

elseif ...

fi;

$Prog_{out}^{D,\pi}$

ensuring $\bigwedge_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P_{int,t_1}^{C_i,\pi} \ \Rightarrow \ ensuring_{int}^{A,\pi} \right) \ \wedge \ ensuring_{out}^{D,\pi}$

Figure 15.9: General form of an internal transition in the expansion of a composite automaton

Chapter 16

Expansion of an example composite automaton

A good preparation takes longer than the delivery.

— E. Kim Nebeuts [93]

In this chapter, we detail the expansion the composite automaton introduced in Example 10.4. In this expansion, we apply the techniques described in Chapter 15 to the composite automaton `Sys` shown in Figure 10.4 and to the canonical versions of its component automata shown in Figures 14.3–14.5. In Section 16.2, we derive the signature of `SysExpanded` in three stages. In Section 16.3, we describe the state of the expanded automaton, including its initial values, and an invariant about the scope of definition for its state variables.

Where convenient, we recapitulate definitions developed in previous sections in summary tables to save the reader (and the authors!) from having to flip back to look up definitions.

16.1 Desugared hidden statement of `Sys`

Following the procedure described in Section 15.2, we eliminate terms other than variable references from the parameters of the **hidden** statement of automaton `Sys` by replacing $params_{hide_1}^{\texttt{Sys,send}} = \langle \texttt{nProcesses, nProcesses+1, x:Int} \rangle$ with $vars^{\texttt{Sys,send}} = \langle \texttt{n1:Int}, \texttt{n2:Int}, \texttt{m:Int} \rangle$, defining σ_1^{hide} to map `m:Int` to a fresh variable `i:Int`, and rewriting the **where** clause in the **hidden** statement to produce

```
hidden send(n1, n2, m)
    where ∃ i:Int (i = m ∧ n1 = nProcesses ∧ n2 = nProcesses+1)
```

which simplifies to

hidden `send(n1, n2, m)` **where** `n1 = nProcesses ∧ n2 = nProcesses+1`

Thus, we define $H^{\texttt{Sys,send}}$ to be `n1 = nProcesses ∧ n2 = nProcesses+1`.

16.2 Signature of SysExpanded

To expand the signature of composite automaton `Sys` as described in Section 15.3, we first calculate the per-kind, per-action, per-component predicates $P_{kind}^{\texttt{Sys},C_i,\pi}$. Then we combine these by component to form the provisional kind predicates $Prov_{kind}^{\texttt{Sys},\pi}$. Finally, we combine these predicates with the **hidden** statement predicate to derive the signature predicates $P_{in}^{\texttt{Sys},\pi}$, $P_{out}^{\texttt{Sys},\pi}$, and $P_{int}^{\texttt{Sys},\pi}$.

In computing these predicates it is helpful to remember the component predicates and canonical variables of the sample composite automaton `Sys`. Table 16.1 collects the former from Example 10.4. Table 16.2 recalls the latter as they were defined in Example 14.2. The local variables shown are derived from Example 14.2 as described in Section 15.6.

PREDICATE	VALUE
$P^{\texttt{Sys,C}}$	`j = i+1 ∧ 1 ≤ i ∧ i < nProcesses`
$P^{\texttt{Sys,P}}$	`1 ≤ n ∧ n ≤ nProcesses`
$P^{\texttt{Sys,W}}$	true

Table 16.1: Component predicates of the sample composite automaton `Sys`

16.2.1 Actions per component

First, we define predicates for each kind of each action for each component. `Sys` has three components and four action names, each of up to three kinds. Thus, there are thirty-six possible per-kind, per-action, per-component predicates $P_{kind}^{\texttt{Sys},C_i,\pi}$. Table 16.3 shows the seven of these predicates that are not trivially false. All the existential quantifiers have been eliminated from the predicates shown in the table.

We can simplify such a predicate by dropping existential quantifiers and conjuncts that are superfluous. A quantifier is superfluous if the predicate equates the quantified variable directly with a term not involving a quantified variable. The conjunct that equates the quantified variable to a defining term is also superfluous. The simplification proceeds in four steps:

1. Define a substitution that maps any superfluous existential variables to the corresponding term.

2. Apply the substitution to the predicate.

3. Delete identity conjuncts from the **where** clause.

CANONICAL SEQUENCE	VARIABLES
$vars^{\texttt{Sys}}$	`nProcesses:Int`
$vars^{\texttt{C}}$	`n:Int`
$vars^{\texttt{P}}$	`n:Int`
$vars^{\texttt{Sys,send}}$	`n1:Int, n2:Int, m:Int`
$vars^{\texttt{Sys,receive}}$	`n1:Int, n2:Int, m:Int`
$vars^{\texttt{Sys,overflow}}$	`i1:Int, s:Set[Int]`
$vars^{\texttt{Sys,found}}$	`i1:Int`
$localVars^{\texttt{Sys,overflow}}$	`P:Map[Int, Locals[P, overflow]]`, `W:Locals[Watch, Int, overflow]`

Table 16.2: Canonical variables used to expand the sample composite automaton `Sys`

PREDICATE	VALUE
$P_{in}^{\texttt{Sys,C,send}}$	`(1` $\leq$ `n1` $\wedge$ `n1` $<$ `nProcesses)` $\wedge$ `(n2` $=$ `n1+1)`
$P_{out}^{\texttt{Sys,P,send}}$	`(1` $\leq$ `n1` $\wedge$ `n1` $\leq$ `nProcesses)` $\wedge$ `(n2` $=$ `n1+1)`
$P_{out}^{\texttt{Sys,C,receive}}$	`(1` $\leq$ `n1` $\wedge$ `n1` $<$ `nProcesses)` $\wedge$ `(n2` $=$ `n1+1)`
$P_{in}^{\texttt{Sys,P,receive}}$	`(1` $\leq$ `n2` $\wedge$ `n2` $\leq$ `nProcesses)` $\wedge$ `(n1` $=$ `n2-1)`
$P_{out}^{\texttt{Sys,P,overflow}}$	`1` $\leq$ `i1` $\wedge$ `i1` $\leq$ `nProcesses`
$P_{in}^{\texttt{Sys,W,overflow}}$	`i1` $\in$ `between(1, nProcesses)`
$P_{out}^{\texttt{Sys,W,found}}$	`i1` $\in$ `between(1, nProcesses)`

Table 16.3: Simplified predicates defining contributions to the signature of `Sys`

4. Delete the existential quantifiers for variables that no longer appear in the predicate.

For example, by the definition given in Section 15.3,

$$
\begin{aligned}
P_{in}^{\texttt{Sys,C,send}} \quad ::= \quad & \exists vars^{\texttt{Sys,C}} \, (P^{\texttt{Sys,C}} \ \wedge \ P_{in}^{\texttt{C,send}}) \\
= \quad & \exists \ \texttt{n:Int} \ (\texttt{1} \leq \texttt{n} \wedge \texttt{n} < \texttt{nProcesses} \wedge \texttt{n1} = \texttt{n} \wedge \texttt{n2} = \texttt{n+1})
\end{aligned}
$$

We simplify this predicate by defining and applying a substitution that maps `n:Int` to `n1:Int`, delete the resulting identity conjunct, the quantified variable, and the quantifier, resulting in the predicate shown in Table 16.3.

PREDICATE	VALUE
$Prov_{in}^{\texttt{Sys,send}}$	$P_{in}^{\texttt{Sys,C,send}}$
$Prov_{out}^{\texttt{Sys,send}}$	$P_{out}^{\texttt{Sys,P,send}}$
$Prov_{out}^{\texttt{Sys,receive}}$	$P_{out}^{\texttt{Sys,C,receive}}$
$Prov_{in}^{\texttt{Sys,receive}}$	$P_{in}^{\texttt{Sys,P,receive}}$
$Prov_{out}^{\texttt{Sys,overflow}}$	$P_{out}^{\texttt{Sys,P,overflow}}$
$Prov_{in}^{\texttt{Sys,overflow}}$	$P_{in}^{\texttt{Sys,W,overflow}}$
$Prov_{out}^{\texttt{Sys,found}}$	$P_{out}^{\texttt{Sys,W,found}}$

Table 16.4: Provisional **where** predicates for the signature of `Sys`

16.2.2 Provisional action kinds

Since no two components of `Sys` share the same kind of any action, it is simple to define the provisional kind predicates $Prov_{kind}^{\texttt{Sys},\pi}$. Seven of the twelve possible predicates are not trivially false. Each of these has exactly one nontrivial disjunct—the corresponding predicate $P_{kind}^{\texttt{Sys},C_i,\pi}$, as shown in Table 16.4

16.2.3 Signature predicates

We now compute the nontrivial signature predicates $P_{in}^{\texttt{Sys},\pi}$, $P_{out}^{\texttt{Sys},\pi}$, and $P_{int}^{\texttt{Sys},\pi}$ for the four action labels `send`, `receive`, `overflow`, and `found` of automaton `SysExpanded`.

Output actions

We compute the signature predicate for output action `send`, by applying the formula

$$P_{out}^{\texttt{Sys,send}} = Prov_{out}^{\texttt{Sys,P,send}} \ \wedge \ \neg H^{\texttt{Sys,send}}.$$

Using the desugared form of the **hidden** predicate shown in Example 15.2, we find that $P_{out}^{\texttt{Sys,send}}$ is

```
1 ≤ n1 ∧ n1 ≤ nProcesses ∧ n2 = n1+1
  ∧ ¬(n1 = nProcesses ∧ n2 = nProcesses+1)
```

Computing the predicates for output actions `receive`, `found`, and `overflow` is simple because there is no **hidden** clause applying to them (*i.e.*, $H^{\texttt{Sys},\pi}$ is false) and the action predicate is, in fact, just the provisional kind predicate, as shown in Figure 16.1.

Input actions

We compute the signature predicate for input action `send` by applying the formula

$$P_{in}^{\texttt{Sys,send}} = Prov_{in}^{\texttt{Sys,send}} \wedge \neg Prov_{out}^{\texttt{Sys,send}}.$$

Thus, $P_{in}^{\texttt{Sys,send}}$ evaluates to

$$\texttt{1} \leq \texttt{n1} \wedge \texttt{n1} < \texttt{nProcesses} \wedge \texttt{n2} = \texttt{n1+1} \wedge \neg((\texttt{1} \leq \texttt{n1} \wedge \texttt{n1} \leq \texttt{nProcesses}) \wedge (\texttt{n2} = \texttt{n1+1}))$$

The signature predicates for input actions `receive`, and `overflow` are computed similarly and appear in Figure 16.1.

Internal actions

In Example 10.4, the component automata have no internal actions. Therefore, the only internal action in `Sys` is the hidden action `send`. Thus, the predicate $P_{int}^{\texttt{Sys,send}}$ is equivalent to

$$Prov_{out}^{\texttt{Sys,send}} \wedge H^{\texttt{Sys,send}},$$

which evaluates to

$$\texttt{1} \leq \texttt{n1} \wedge \texttt{n1} \leq \texttt{nProcesses} \wedge \texttt{n2} = \texttt{n1+1} \wedge \texttt{n1=nProcesses} \wedge \texttt{n2=nProcesses+1}$$

The complete expanded signature of automaton `Sys` is given in Figure 16.1.

16.3 States and **initially** predicates of SysExpanded

The complete expanded state of automaton `Sys` is given in Figure 16.1. Since each component of the desugared composite automaton has non-type parameters, all three state variables are maps. Three of the **initially** subclauses (and the subsequent invariant) assert the well-formedness requirement that each map is defined only for values of the component parameters on which the component itself is defined. The other three **initially** subclauses assert that the `contents` of each `channel` is initially empty, the `watch` process is looking for values between 1 and `nProcesses` and that each process `P` initially has value 0 and nothing to send. The **type** declaration appearing at the beginning of the figure is the automatically generated sort for the state of the composite automaton.

16.4 Input Transition Definitions of SysExpanded

We compute the input transitions of `SysExpanded` by following the pattern of Figure 15.3 for each of the input actions in its signature (`receive`, `send`, and `overflow`) and simplifying. Figure 16.2 shows

```
type States[Sys] = tuple of C:Map[Int, States[Channel,Int,Int]],
                            P:Map[Int, States[P]],
                            W:States[Watch,Int]

automaton SysExpanded(nProcesses:Int)
  signature
    output send(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 ≤ nProcesses ∧ n2 = n1+1
                      ∧ ¬(n1 = nProcesses ∧ n2 = nProcesses+1),
           receive(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1,
           overflow(i1:Int, s:Set[Int]) where 1 ≤ i1 ∧ i1 ≤ nProcesses,
           found(i1:Int) where i1 ∈ between(1, nProcesses)
    input send(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1
                     ∧ ¬(1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1),
          receive(n1, n2, m:Int)
             where 1 ≤ n2 ∧ n2 ≤ nProcesses ∧  n1 = n2-1
                   ∧  ¬(1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1),
          overflow(i1:Int, s:Set[Int])
             where i1 ∈ between(1, nProcesses)
                     ∧ ¬(1 ≤ i1 ∧ i1 ≤ nProcesses)
    internal send(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 ≤ nProcesses ∧ n2 = n1+1
                      ∧ n1 = nProcesses ∧ n2 = nProcesses+1
  states C:Map[Int, States[Channel, Int, Int]],
         P:Map[Int, States[P]],
         W:States[Watch, Int]
    initially
        ∀ n:Int ((1 ≤ n ∧ n < nProcesses) ⇒ C[n].contents = {})
      ∧ ∀ n:Int ((1 ≤ n ∧ n < nProcesses) ⇔ defined(C, n))
      ∧ ∀ n:Int ((1 ≤ n ∧ n ≤ nProcesses) ⇒ P[n].val = 0 ∧ P[n].toSend = {})
      ∧ ∀ n:Int ((1 ≤ n ∧ n ≤ nProcesses) ⇔ defined(P, n))
      ∧ W.seen = constant(false)
  ...
invariant of SysExpanded:
  ∀ n:Int (1 ≤ n ∧ n < nProcesses ⇔ defined(C[n]));
  ∀ n:Int (1 ≤ n ∧ n ≤ nProcesses ⇔ defined(P[n]))
```

Figure 16.1: Expanded signature and states of the sample composite automaton `Sys`

the three resulting forms.

In that figure, each input transition is formed from only a single contributing component. Thus, the conjunctions in the **where** over the contributing components in Figure 15.3 each resolves to a single term. Furthermore, we omit the **where** clauses for the `receive` and `send` transitions because the transition definitions have no local variables. In each of the three transitions, we omit the **ensuring** predicate altogether because the sole contributing predicate for each transition ($ensuring_{in}^{\texttt{P},\texttt{receive}}$, $ensuring_{in}^{\texttt{C},\texttt{send}}$, and $ensuring_{in}^{\texttt{W},\texttt{overflow}}$) is trivially true. The **eff** clause of each transition resolves to a single **for** loop or conditional. In the `overflow` transition, the conditional is replaced by its body because there is only a single contributing transition.

Figure 16.3 shows the final text of the expanded input transitions. In that figure, we omit the local variable `P:Map[Int, Locals[P, overflow]]` from the `overflow` transition because it does not appear in the transition precondition or effects. The **where** clause predicate $P_{in,t_1}^{\texttt{Sys},\texttt{W},\texttt{overflow}}$ reduces to the implication shown in Table 16.5 because $vars^{\texttt{Sys},\texttt{W}}$ is empty and $P^{\texttt{Sys},\texttt{W}}$ is trivially true.

The **for** loops in the `receive` and `send` transitions have been eliminated by the following simplification. Filling in the specified variables from Tables 16.2, predicates from Tables 16.1 and 16.5 and statements from Example 14.2 in the `receive` transition **for** loop yields the loop

```
for n:Int where (1 ≤ n ∧ n ≤ nProcesses ∧ n1 = n-1 ∧ n2 = n) do
  if P[n].val = 0 then P[n].val := m
  elseif m < P[n2].val then
    P[n].toSend := insert(P[n].val, P[n].toSend);
    P[n].val := m
  elseif P[n].val < m then
    P[n].toSend := insert(m, P[n].toSend)
  fi
od.
```

Since the last conjunct of the loop **where** clause limits the loop variable to a single value, the transition parameter `n2`, we can eliminate the loop altogether. Thus, in Figure 16.3, we replace the loop with its body after applying to the body a substitution that maps the loop variable `n` to its value `n2`. Similarly, the **for** loop in the `send` transition is eliminated using a substitution that maps its loop variable `n` to the transition parameter `n1`.

16.5 Output Transition Definitions of `SysExpanded`

We compute the output transitions of `SysExpanded` by following the pattern of Figure 15.7 for each of the output actions in its signature (`receive`, `send`, `overflow`, and `found`) and simplifying. Figure 16.4 shows the four resulting forms.

PREDICATE	VALUE
$P_{in}^{\texttt{P,receive}}$	`n1 = n-1 ∧ n2 = n`
$P_{in}^{\texttt{C,send}}$	`n1 = n ∧ n2 = n+1`
$P_{in}^{\texttt{W,overflow}}$	`i1 ∈ between(1, nProcesses)`
$P_{in,t_1}^{\texttt{W,overflow}}$	`s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s)`
$P_{in,t_1}^{\texttt{Sys,W,overflow}}$	`i1 ∈ between(1, nProcesses) ⇒ (s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s))`

Table 16.5: Nontrivial predicates used in expanding input transition definitions of the sample composite automaton `Sys` derived from Figures 14.3, 14.4 and 14.5

input $\texttt{receive}(vars^{\texttt{Sys,receive}})$
  eff for $vars^{\texttt{Sys,P}}$ **where** $P^{\texttt{Sys,P}} \wedge P_{in}^{\texttt{P,receive}}$ **do** $Prog_{in}^{\texttt{P,receive}}$ **od**

input $\texttt{send}(vars^{\texttt{Sys,send}})$
  eff for $vars^{\texttt{Sys,C}}$ **where** $P^{\texttt{Sys,C}} \wedge P_{in}^{\texttt{C,send}}$ **do** $Prog_{in}^{\texttt{C,send}}$ **od**

input $\texttt{overflow}(vars^{\texttt{Sys,overflow}};$ **local** $localVars^{\texttt{Sys,overflow}})$ **where** $P_{in,t_1}^{\texttt{Sys,W,overflow}}$
  eff $Prog_{in}^{\texttt{W,overflow}}$

Figure 16.2: Form of input transitions of `SysExpanded`

```
input receive(n1, n2, m)
  eff if P[n2].val = 0 then P[n2].val := m
      elseif m < P[n2].val then
        P[n2].toSend := insert(P[n2].val, P[n2].toSend);
        P[n2].val := m
      elseif P[n2].val < m then
        P[n2].toSend := insert(m, P[n2].toSend)
      fi

input send(n1, n2, m)
  eff C[n1].contents := insert(m, C[n1].contents)

input overflow(i,s; locals W:Locals[W,int,overflow])
  where i1 ∈ between(1, nProcesses) ⇒ (s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s))
  eff if s = W.s2 ∪ {i1} then W.seen[i1] := true
      elseif ¬(i1 ∈ s) then W.seen[i1] := false
      fi
```

Figure 16.3: Input transition definitions of `SysExpanded`

PREDICATE	VALUE
$P_{out}^{\texttt{C,receive}}$	`n1 = n ∧ n2 = n+1`
$P_{out,t_1}^{\texttt{C,receive}}$	`n1 = n ∧ n2 = n+1`
$Pre_{out}^{\texttt{C,receive}}$	`m ∈ C[n].contents`
$P_{out}^{\texttt{P,send}}$	`n1 = n ∧ n2 = n+1`
$P_{out,t_1}^{\texttt{P,send}}$	`n1 = n ∧ n2 = n+1`
$Pre_{out}^{\texttt{P,send}}$	`m ∈ P[n].toSend`
$P_{out}^{\texttt{P,overflow}}$	`i1 = n`
$P_{out,t_1}^{\texttt{P,overflow}}$	`i1 = n`
$Pre_{out}^{\texttt{P,overflow}}$	`s = P[n].toSend ∧ n < size(s) ∧ P[n].t ⊆ s`
$P_{out}^{\texttt{W,found}}$	`i1 ∈ between(1, nProcesses)`
$Pre_{out}^{\texttt{W,found}}$	`W.seen[i1]`

Table 16.6: Nontrivial predicates used in expanding output transition definitions of the sample composite automaton `Sys` derived from Figures 14.3, 14.4 and 14.5

Notice that only one component contributes an output transition to each expanded output transition. Therefore, only syntactic elements from the sole contributing component and the corresponding expanded input action appear in each transition. Each local variable list contains of the component variables for that contributing component. Since, $localVars^{\texttt{Sys,receive}}$, $localVars^{\texttt{Sys,send}}$, and $localVars^{\texttt{Sys,found}}$ are empty, they are omitted from their respective transitions. Since component `W` is unparameterized, the `found` transition has no local variables at all.

The **where** clause of each transition resolves to a single term rather than being a disjunction over the contributing components. Furthermore, we omit the **where** clauses for the `receive`, `send`, and `found` transitions because the transition definitions have no local variables. Similarly, the ensuring clause is only a single conjunction. In each of the four transitions, we omit the **ensuring** predicate altogether because the consequent for each transition ($ensuring_{out}^{\texttt{C,receive}}$, $ensuring_{out}^{\texttt{P,send}}$, $ensuring_{out}^{\texttt{P,overflow}}$, and $ensuring_{out}^{\texttt{W,found}}$) is trivially true. Furthermore, the conditional and guarding conjunction can be omitted from the **eff** clause because only one output contributes. So each effect is just the effect of the contributing output transition followed by the effect of the corresponding expanded input transition. Since the output transition definition for the `found` action in component `W` has no effect and there is no `found` input action, the expanded found transition has no effect either.

Filling in the specified variables from Tables 16.2, predicates from Tables 16.1 and 16.6 and statements from Example 14.2 and Figure 16.3 yields the complete the complete text of the expanded output transitions shown in Figure 16.5. We simplify the transition definitions using two techniques. First, we eliminating unneeded local variables. Second, we use the fact that the signature **where**

predicate for an action (*e.g.*, $P_{out}^{\texttt{Sys,receive}}$) is implicitly conjoined to the corresponding transition **where** predicate (*e.g.*, $P_{out,t_1}^{\texttt{Sys,receive}}$) and precondition ($Pre_{out}^{\texttt{Sys,receive}}$) to eliminate redundant assertions in the transition **where** predicate and precondition. The resulting final form of output transitions is shown in Figure 16.6.

To eliminate unneeded local variables, we follow the four step process to eliminate unneeded local variables described in Section 12.2. For example, we note that the **where** clause of the `receive` transiting equates `n` with parameter `n1`. Furthermore, there is no assignment to `n` in the effects of that transition. Thus, the local variable `n` is extraneous. So, we define a substitution that maps the local variable `n` to the parameter `n1` and apply it to the **where**, **pre**, and **eff** clauses. We then delete the resulting identity conjunct from the **where** clause and the declaration of the local variable `n`. Similarly simplifications eliminate the local variable `n` from the `send` and `overflow` transition definitions. Since the resulting `receive` and `send` transitions no longer have any local variables, we omit their **where** clauses altogether.

After this simplification, the precondition for the `receive` transition is

pre `1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1 ∧ m ∈ C[n1].contents`

However, the first three conjuncts are also asserted by the the signature **where** clause for the `receive` output action $P_{out}^{\texttt{Sys,receive}}$ and, therefore, are redundant. Similarly simplifications to the **where** and **pre** clauses of the other transitions result in the final text of the expanded output transitions shown in Figure 16.6.

16.6 Internal Transition Definitions of `SysExpanded`

Since no component has any internal transitions, the only internal transitions in `SysExpanded` is the hidden output `send` transitions. In the case where no component contributes an internal transition, the form in Figure 15.9 reduces exactly that in Figure 15.7. That is, the internal `send` transition definition is identical to the output transition definition except for its label. The two actions are distinguished exactly by the assertion or negation of $H^{\texttt{Sys,send}}$ in the signature of `SysExpanded`. The final transition of `SysExpanded` is shown in Figure 16.7.

output $\texttt{receive}(vars^{\texttt{Sys},\texttt{receive}};\ \textbf{local}\ vars^{\texttt{Sys},\texttt{C}})$
  where $P^{\texttt{Sys},\texttt{C}} \wedge P_{out}^{\texttt{C},\texttt{receive}} \wedge P_{out,t_1}^{\texttt{C},\texttt{receive}} \wedge P_{in,t_1}^{\texttt{Sys},\texttt{receive}}$
  pre $P^{\texttt{Sys},\texttt{C}} \wedge P_{out}^{\texttt{C},\texttt{receive}} \wedge Pre_{out}^{\texttt{C},\texttt{receive}}$
  eff $Prog_{out}^{\texttt{C},\texttt{receive}};$
    $Prog_{in}^{\texttt{Sys},\texttt{receive}}$

output $\texttt{send}(vars^{\texttt{Sys},\texttt{send}};\ \textbf{local}\ vars^{\texttt{Sys},\texttt{P}})$
  where $P^{\texttt{Sys},\texttt{P}} \wedge P_{out}^{\texttt{P},\texttt{send}} \wedge P_{out,t_1}^{\texttt{P},\texttt{send}} \wedge P_{in,t_1}^{\texttt{Sys},\texttt{send}}$
  pre $P^{\texttt{Sys},\texttt{P}} \wedge P_{out}^{\texttt{P},\texttt{send}} \wedge Pre_{out}^{\texttt{P},\texttt{send}}$
  eff $Prog_{out}^{\texttt{P},\texttt{send}};$
    $Prog_{in}^{\texttt{Sys},\texttt{send}}$

output $\texttt{overflow}(vars^{\texttt{Sys},\texttt{overflow}};\ \textbf{local}\ vars^{\texttt{Sys},\texttt{C}}, vars^{\texttt{Sys},\texttt{P}}, localVars^{\texttt{Sys},\texttt{overflow}})$
  where $P^{\texttt{Sys},\texttt{P}} \wedge P_{out}^{\texttt{P},\texttt{overflow}} \wedge P_{out,t_1}^{\texttt{P},\texttt{overflow}} \wedge P_{in,t_1}^{\texttt{Sys},\texttt{overflow}}$
  pre $P^{\texttt{Sys},\texttt{P}} \wedge P_{out}^{\texttt{P},\texttt{overflow}} \wedge Pre_{out}^{\texttt{P},\texttt{overflow}}$
  eff $Prog_{out}^{\texttt{P},\texttt{overflow}};$
    $Prog_{in}^{\texttt{Sys},\texttt{overflow}}$

output $\texttt{found}(vars^{\texttt{Sys},\texttt{found}})$
  pre $P^{\texttt{Sys},\texttt{W}} \wedge P_{out}^{\texttt{W},\texttt{found}} \wedge Pre_{out}^{\texttt{W},\texttt{found}}$

Figure 16.4: Form of output transitions of `SysExpanded`

```
output receive(n1, n2, m; local n:Int)
       where 1 ≤ n1 ∧ n1 < nProcesses ∧ n1 = n ∧ n2 = n1+1
  pre 1 ≤ n ∧ n1 < nProcesses ∧ n1 = n ∧ n2 = n+1
      ∧ m ∈ C[n].contents
  eff
    C[n].contents := delete(m, C[n].contents)
    if P[n2].val = 0 then P[n2].val := m
    elseif m < P[n2].val then
      P[n2].toSend := insert(P[n2].val, P[n2].toSend);
      P[n2].val := m
    elseif P[n2].val < m then
      P[n2].toSend := insert(m, P[n2].toSend)
    fi

output send(n1, n2, m; local n:Int)
         where 1 ≤ n ∧ n ≤ nProcesses ∧ n1 = n ∧ n2 = n+1
               ∧ n1 = n ∧ n2 = n+1
   pre 1 ≤ n ∧ n ≤ nProcesses ∧ n1 = n ∧ n2 = n+1 ∧ m ∈ P[n].toSend
   eff
     P[n].toSend := delete(m, P[n].toSend)
     C[n1].contents := insert(m, C[n1].contents)

output overflow(i1, s; local n:Int,
                           P:Map[Int, Locals[P, overflow]],
                           W:Locals[Watch, Int, overflow])
   where 1 ≤ n ∧ n ≤ nProcesses ∧ i1 = n ∧
         i1 ∈ between(1, nProcesses) ⇒ (s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s))
   pre 1 ≤ n ∧ n ≤ nProcesses ∧ i1 = n ∧
       s = P[n].toSend ∧ n < size(s) ∧ P[n].t ⊆ s
   eff P[n].toSend := P[n].t
       if s = W.s2 ∪ {i1} then W.seen[i1] := true
       elseif ¬(i1 ∈ s) then W.seen[i1] := false
       fi

 output found(i1)
   pre i1 ∈ between(1, nProcesses) ∧ W.seen[i1]
```

Figure 16.5: Output transition definitions of `SysExpanded`

```
output receive(n1, n2, m)
  pre m ∈ C[n1].contents
  eff
    C[n1].contents := delete(m, C[n1].contents)
    if P[n2].val = 0 then P[n2].val := m
    elseif m < P[n2].val then
      P[n2].toSend := insert(P[n2].val, P[n2].toSend);
      P[n2].val := m
    elseif P[n2].val < m then
      P[n2].toSend := insert(m, P[n2].toSend)
    fi

output send(n1, n2, m)
   pre m ∈ P[n1].toSend
   eff
     P[n1].toSend := delete(m, P[n1].toSend)
     C[n1].contents := insert(m, C[n1].contents)

output overflow(i1, s; local P:Map[Int, Locals[P, overflow],
                            W:Locals[Watch, Int, overflow])
   where s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s)
   pre s = P[i1].toSend ∧ i1 < size(s) ∧ P[i1].t ⊆ s
   eff P[i1].toSend := P[i1].t
       if s = W.s2 ∪ {i1} then W.seen[i1] := true
       elseif ¬(i1 ∈ s) then W.seen[i1] := false
       fi

 output found(i1)
   pre W.seen[i1]
```

Figure 16.6: Simplified output transition definitions of `SysExpanded`

```
internal send(n1, n2, m)
   pre m ∈ P[n1].toSend
   eff
     P[n1].toSend := delete(m, P[n1].toSend)
     C[n1].contents := insert(m, C[n1].contents)
```

Figure 16.7: Internal transition definitions of `SysExpanded`

Chapter 17

Renamings, Resortings, and Substitutions

One man's constant is another man's variable.

— Alan J. Perlis [99]

In this section, we give formal definitions for resortings and variable substitutions in IOA.

17.1 Sort renamings

A *sort renaming* or *resorting* is a map from simple sorts to sorts.[1] Any resorting ρ extends naturally to a map $\dot{\rho}$ defined for all simple sorts by letting $\dot{\rho}$ be the identity on elements not in the domain of ρ. In turn, $\dot{\rho}$ extends further to a map $\ddot{\rho}$ from sorts to sorts by the following recursive definition:

$$\ddot{\rho}(u) ::= \begin{cases} \dot{\rho}(T) & \text{if } u \text{ is a simple sort } T, \text{ and} \\ T[\ddot{\rho}(T_1), \ldots, \ddot{\rho}(T_n)] & \text{if } u \text{ is a compound sort } T[T_1, \ldots, T_n]. \end{cases}$$

Let $\rho_{S \to T}$ denote a resorting that maps the sort S to sort T and is otherwise the same as ρ (even if S is already in the domain of ρ).

[1] In IOA, sorts are divided into *simple* or *primitive* sorts, such as `Int` and `T`, and *compound* or *constructed* sorts, such as `Set[T]` and `WeightedGraph[Node,Nat]`.

17.2 Variable renamings

A *variable renaming* ρ_q is an extension of a resorting ρ that maps variables in a sequence q to distinct variables. If v is a variable $i{:}T$ in q, then $\rho_q(v)$ is defined to be $j{:}\ddot{\rho}(T)$ where j is an identifier (i itself, if possible) such that $j{:}\ddot{\rho}(T) \neq \rho_q(v')$ for all variables v' that precede v in q. We say that ρ_q is a *variable renaming with respect to precedence sequence* q.

If ρ_r is a variable renaming where $r = q \| p$ then we say ρ_r is an *extension of* ρ_q *with respect to precedence sequence* p and we write that $\rho_r = \rho_q \vdash p$.

17.3 Operator renamings

An *operator renaming* ω is a map from operators to operators that preserves signatures. Any operator renaming ω extends naturally to a map $\dot{\omega}$ defined for all operators by letting $\dot{\omega}$ map each operator not in the domain of ω to itself.

We extend any operator renaming ω further to a map $\ddot{\omega}$ on some syntactic elements of an IOA automaton (terms to terms, statements to statements, etc.) We now define $\ddot{\omega}$ for each type of IOA syntax to which it may apply.

17.3.1 Terms and sequences of terms

If u is a term, then $\ddot{\omega}(u)$ is

- v, if u is a variable v,
- $\dot{\omega}(f)(\ddot{\omega}(u_1), \ldots, \ddot{\omega}(u_n))$, if u is a term $f(u_1, \ldots, u_n)$ for some operator f and terms $u_1, \ldots, u_n$,
- $\forall v\, \ddot{\omega}(u')$, if u is a term $\forall v\, (u')$ for some variable v and term u', and
- $\exists v\, \ddot{\omega}(u')$, if u is a term $\exists v\, (u')$ for some variable v and term u'.

If q is a sequence of terms $\{u_1, u_2, \ldots, u_n\}$, then $\ddot{\omega}(q)$ is $\{\ddot{\omega}(u_1), \ddot{\omega}(u_2), \ldots, \ddot{\omega}(u_n)\}$.

17.3.2 Values

If l is a value, then $\ddot{\omega}(l)$ is

- $\ddot{\omega}(t)$, if l is a term t
- **choose** v **where** $\ddot{\omega}(p)$, if l is a choice **choose** v **where** p for some variable v and predicate p.

17.3.3 Statements and programs

If s is a statement, then $\ddot{\omega}(s)$ is

- $\ddot{\omega}(lhs) := \ddot{\omega}(rhs)$, if s is an assignment $lhs := rhs$ for some lvalue lhs and some value rhs,
- **if** $\ddot{\omega}(p_1)$ **then** $\ddot{\omega}(s_1)$ **elseif** $\ddot{\omega}(p_2)$ **then** ... **else** $\ddot{\omega}(s_n)$ **fi**, if s is a conditional statement **if** p_1 **then** s_1 **elseif** p_2 ... **else** s_n **fi** for some predicates $p_1, \ldots, p_{n-1}$ and statements $s_1, \ldots, s_n$, and
- **for** v **where** $\ddot{\omega}(p)$ **do** $\ddot{\omega}(g)$ **od**, if s is a loop statement **for** v **where** p **do** g **od** for some variable v, predicate p, and program g.

If g is a program $s_1; s_2; \ldots$, then $\ddot{\omega}(g)$ is $\ddot{\omega}(s_1); \ddot{\omega}(s_2); \ldots$.

17.3.4 Shorthand tuple sort declarations

If ω is an operator renaming and d_1 and d_2 are two shorthand **tuple** sort declarations:

$$d_1 \quad ::= \quad T \textbf{ tuple of } i_1{:}T_1, i_2{:}T_2, \ldots, \text{and}$$
$$d_2 \quad ::= \quad T \textbf{ tuple of } j_1{:}T_1, j_2{:}T_2, \ldots,$$

where i_1, $i_2, \ldots$, and j_1, $j_2, \ldots$, are identifiers and T, T_1, $T_2, \ldots$, are sorts then we write $\omega_{d_1 \to d_2}$ or $\omega_{T,\{i_1,i2,\ldots\} \to \{j_1,j_2,\ldots\}}$ for the operator renaming that maps

1. tuple selection operators $\_\_.i_k{:}T \to T_k$ to $\_\_.j_k{:}T \to T_k$, and
2. tuple set operators $\texttt{set\_}i_k{:}T, T_k \to T$ to $\texttt{set\_}j_k{:}T, T_k \to T$.

17.4 Renamings for automata

In Chapter 13 we defined resortings that map $types^A$ to $actualTypes^{D,A}$ for some desugared automaton A with formal type parameters $types^A$ instantiated with actual type parameters $actualTypes^{D,A}$.

Let ρ be such a resorting and ϱ be the variable renaming $\rho_{\{\}}$. We extend ϱ to a map $\dot{\varrho}$ on some syntactic elements of an IOA automaton (terms to terms, statements to statements, etc.) by defining $\dot{\varrho}$ for each type of IOA syntax to which it may apply.

17.4.1 Automata

If A is desugared primitive automaton with syntax as given in Chapter 12 and shown in Figure 12.5, then $\dot{\varrho}(A)$ is[2]

[2]Strictly speaking, the definition of the automaton $\dot{\varrho}(A)$ is not a legal definition of a primitive IOA automaton. Its type parameters, shown as $types^A$, should really consist of the non-built-in types that appear in sorts in $\dot{\varrho}(types^A)$. Furthermore, the declared state variables may not match the aggregate state variable selectors that appear in terms in signature **where** clauses, in the **initially** clause, or in transition definitions.

automaton $A(\dot{\varrho}^A(vars^A); types^A)$

signature

...

kind $\pi(\dot{\varrho}^{A,\pi}(vars^{A,\pi}))$ **where** $\dot{\varrho}^{A,\pi}(P^{A,\pi}_{kind})$

...

states $\rho(stateVars^A) := \dot{\varrho}^A(initVals^A)$ **initially** $\dot{\varrho}^A(P^A_{init})$

transitions

...

$$\dot{\varrho}^{A,\pi}_{kind,t_1}\left[\begin{array}{l} \textbf{kind}\ \pi(vars^{A,\pi};\ \textbf{local}\ localVars^{A,\pi}_{kind})\ \textbf{where}\ P^{A,\pi}_{kind,t_1} \\ \quad \textbf{pre}\ Pre^{A,\pi}_{kind,t_1} \\ \quad \textbf{eff}\ Prog^{A,\pi}_{kind,t_1}\ \textbf{ensuring}\ ensuring^{A,\pi}_{kind,t_1} \end{array}\right]$$

....

where

1. $\dot{\varrho}^A$ is a variable renaming $\dot{\varrho} \vdash (\{A, A'{:}States[A, types^A]\} \| \ vars^A \| \ stateVars^A \| \ postVars^A)$.[3]

2. $\dot{\varrho}^{A,\pi}$ is a variable renaming $\dot{\varrho}^A \vdash vars^{A,\pi}$.

3. $\dot{\varrho}^{A,\pi}_{kind,t_1}$ is a variable renaming
   $\dot{\varrho}^{A,\pi} \vdash (\{A, A'{:}Locals[A, types^A, kind, \pi]\} \| \ localVars^{A,\pi}_{kind} \| \ localPostVars^{A,\pi}_{kind})$.[4]

[3]Even though variables in $stateVars^A$ and $postVars^A$ do not appear in any terms in a desugared automaton definition, we include those variables in the precedence sequence to ensure that selectors for local variables do not clash with selectors for state variables in transition definitions (see below).

[4]Like state variables, variables in $localVars^{A,\pi}_{kind}$ and $localPostVars^{A,\pi}_{kind}$ do not appear in any terms in a desugared automaton definition. We include those variables in the precedence sequence only to ensure that selectors for local variables do not clash with each other. (see below).

17.4.2 Transition definitions

Let t be a transition definition in automaton A as given above. That is, t is

kind $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi}_{kind})$ **case** c **where** p_1

pre p_2

eff g **ensuring** p_3

where $vars^{A,\pi}$ is a sequence of variables, $localVars^{A,\pi}_{kind} = \{i_1{:}T_1, i_2{:}T_2, \ldots, \}$ is a sequence of variables, p_1, p_2, and p_3 are predicates, and g is a program. Let S be the aggregate local sort $Locals[A, types^A, kind, \pi]$ of t, and $\dot{\varrho}$ be the variable renaming $\dot{\varrho}^{A,\pi}_{kind,t_l}$ given above. That is, $\dot{\varrho}$ is an extension of ρ with respect to the precedence sequence $\{A, A'{:}States[A, types^A]\} \| \ vars^A \| \ stateVars^A \| \ postVars^A \| \ vars^{A,\pi} \| \ \{A, A'{:}Locals[A, types^A, kind, \pi]\} \| \ localVars^{A,\pi}_{kind} \| \ localPostVars^{A,\pi}_{kind}$.
We define $\dot{\varrho}(t)$ to be

kind $\pi(\dot{\varrho}(vars^{A,\pi});\ \dot{\varrho}(localVars^{A,\pi}_{kind}))$ **case** c **where** $\ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(p_1))$

pre $\ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(p_2))$

eff $\ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(g))$ **ensuring** $\ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(p_3))$.

where $\dot{\varrho}(localVars^{A,\pi}_{kind})$ is a variable sequence $\{j_1{:}\rho(T_1), j_2{:}\rho(T_2), \ldots, \}$. Note that if $localVars^{A,\pi}_{kind} = \dot{\varrho}(localVars^{A,\pi}_{kind})$, then $\omega_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}$ is the identity operator renaming.

17.4.3 Statements and programs

If s is a statement and ϱ is some variable renaming, then $\dot{\varrho}(s)$ is

- $\dot{\varrho}(lhs) := \dot{\varrho}(rhs)$, if s is an assignment $lhs := rhs$ for lvalue lhs and value rhs,
- **if** $\dot{\varrho}(p_1)$ **then** $\dot{\varrho}(s_1)$ **elseif** $\dot{\varrho}(p_2)$ **then** $\ldots$ **else** $\dot{\varrho}(s_n)$ **fi**, if s is a conditional statement **if** p_1 **then** s_1 **elseif** $p_2 \ldots$ **else** s_n for some predicates $p_1, \ldots, p_{n-1}$and statements $s_1, \ldots, s_n$, and
- **for** $\dot{\varrho}'(v)$ **where** $\dot{\varrho}'(p)$**do** $\dot{\varrho}'(g)$ **od**, if s is a loop **for** v **where** p **do** g **od** for some variable v, predicate p, and program g, where $\dot{\varrho}' = \dot{\varrho} \vdash \{v\}$.

If g is a program $s_1; s_2; \ldots,$ then $\dot{\varrho}(g)$ is $\dot{\varrho}(s_1); \dot{\varrho}(s_2); \ldots$.

17.4.4 Values

If l is a value and ϱ is some variable renaming, then $\dot{\varrho}(l)$ is

- $\dot{\varrho}(t)$, if l is a term t, and
- **choose** $\dot{\varrho}'(v)$ **where** $\dot{\varrho}'(p)$, if l is a choice **choose** v **where** p for some variable v and predicate p, where $\dot{\varrho}' = \dot{\varrho} \vdash \{v\}$.

17.4.5 Terms and sequences of terms

If u is a term and ϱ is some variable renaming, then $\dot{\varrho}(u)$ is

- $\varrho(v)$, if u is a variable v,
- $f(\dot{\varrho}(u_1), \ldots, \dot{\varrho}(u_n))$, if u is a term $f(u_1, \ldots, u_n)$ for some operator f and terms $u_1, \ldots, u_n$,
- $\forall \dot{\varrho}'(v)\ \dot{\varrho}'(u')$, if u is a term $\forall v\ (u')$ for some variable v and term u', where $\varrho' = \varrho \vdash \{v\}$, and
- $\exists \dot{\varrho}'(v)\ \dot{\varrho}'(u')$, if u is a term $\exists v\ (u')$ for some variable v and term u', where $\varrho' = \varrho \vdash \{v\}$.

If q is a sequence of terms $\{u_1, u_2, \ldots, u_n\}$, then $\dot{\varrho}(q)$ is $\{\dot{\varrho}(u_1), \dot{\varrho}(u_2), \ldots, \dot{\varrho}(u_n)\}$.

17.5 Substitutions

A *substitution* is a map from variables to terms such that the image of any variable has the same sort as the variable. Any substitution σ extends naturally to a map $\dot{\sigma}$ defined for all variables by letting $\dot{\sigma}$ map each variable not in the domain of σ to a term that is a simple reference to the variable itself.

Let $\sigma_{v \to t}$ denote a substitution that maps the variable v to the term t and is otherwise the same as σ (even if v is already in the domain of σ). We extend any substitution σ further to a map $\ddot{\sigma}$ on some syntactic elements of an IOA automaton (terms to terms, statements to statements, etc.). We now define $\ddot{\sigma}$ for each type of IOA syntax to which it may apply.

17.5.1 Terms and sequences of terms

If u is a term, then $\ddot{\sigma}(u)$ is

- $\dot{\sigma}(v)$, if u is a variable v,
- $f(\ddot{\sigma}(u_1), \ldots, \ddot{\sigma}(u_n))$, if u is a term $f(u_1, \ldots, u_n)$ for some operator f and terms $u_1, \ldots, u_n$,
- $\forall w\ \ddot{\sigma}_{v \to w}(u')$, if u is a term $\forall v\ (u')$ for some variable v and term u', where w is a variable (v itself, if possible) with the same sort as v, where $w \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(u)$, and
- $\exists w\ \ddot{\sigma}_{v \to w}(u')$, if u is a term $\exists v\ (u')$ for some variable v and term u', where w is as above.

If q is a sequence of terms $\{u_1, u_2, \ldots, u_n\}$, then $\ddot{\sigma}(q)$ is $\{\ddot{\sigma}(u_1), \ddot{\sigma}(u_2), \ldots, \ddot{\sigma}(u_n)\}$.

17.5.2 Values

If l is a value, then $\ddot{\sigma}(l)$ is

- $\ddot{\sigma}(t)$, if l is a term t

- **choose** w **where** $\ddot{\sigma}_{v \to w}(p)$, if l is a choice **choose** v **where** p for some variable v and predicate p, where w is a variable (v itself, if possible) with the same sort as v, and where $w \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(l)$.

17.5.3 Statements and programs

If s is a statement, then $\ddot{\sigma}(s)$ is

- $\ddot{\sigma}(\mathit{lhs}) := \ddot{\sigma}(\mathit{rhs})$, if s is an assignment $\mathit{lhs} := \mathit{rhs}$ for some lvalue lhs and some value rhs,

- **if** $\ddot{\sigma}(p_1)$ **then** $\ddot{\sigma}(s_1)$ **elseif** $\ddot{\sigma}(p_2)$ **then** $\dots$ **else** $\ddot{\sigma}(s_n)$ **fi**, if s is a conditional statement **if** p_1 **then** s_1 **elseif** $p_2 \dots$ **else** s_n **fi** for some predicates $p_1, \dots, p_{n-1}$ and statements $s_1, \dots, s_n$,

- **for** w **where** $\ddot{\sigma}_{v \to w}(p)$ **do** $\ddot{\sigma}_{v \to w}(g)$ **od**, if s is a loop statement **for** v **where** p **do** g **od** for some variable v, predicate p, and program g, where w is a variable (v itself, if possible) with the same sort as v, where $w \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(s)$.

If g is a program $s_1; s_2; \dots$, then $\ddot{\sigma}(g)$ is $\ddot{\sigma}(s_1); \ddot{\sigma}(s_2); \dots$.

17.5.4 Transition definitions

If, in automaton A parameterized by type parameters types^A, t is a transition definition

kind $\pi(\mathit{params}^\pi;$ **local** $v_1, v_2, \dots)$ **case** c **where** p_1

pre p_2

eff g **ensuring** p_3

where $params^\pi$ is a sequence of terms, $v_1, v_2, \ldots$ is a sequences of variables $i_1{:}T_1, i_2{:}T_2, \ldots$, p_1, p_2, and p_3 are predicates, g is a program, and S is the aggregate local sort of t, then $\ddot{\sigma}(t)$ is

$$\begin{array}{l}
\textbf{kind}\ \pi(\ddot{\sigma}_{\{v_1,v_2,\ldots\}\to\{w_1,w_2,\ldots\}}(params^\pi);\ \textbf{local}\ w_1, w_2, \ldots) \\
\qquad \textbf{case}\ c\ \textbf{where}\ \ddot{\omega}_{S,\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\ddot{\sigma}_{\{v_1,v_2,\ldots\}\to\{w_1,w_2,\ldots\}}(p_1))) \\
\quad \textbf{pre}\ \ddot{\omega}_{S,\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\ddot{\sigma}_{\{v_1,v_2,\ldots\}\to\{w_1,w_2,\ldots\}}(p_2))) \\
\quad \textbf{eff}\ \ddot{\omega}_{S,\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\ddot{\sigma}_{\{v_1,v_2,\ldots\}\to\{w_1,w_2,\ldots\}}(g))) \\
\qquad \textbf{ensuring}\ \ddot{\omega}_{S,\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\ddot{\sigma}_{\{v_1,v_2,\ldots\}\to\{w_1,w_2,\ldots\}}(p_3)))
\end{array}$$

where

1. w_k is a variable $j_k{:}T_k$ (v_k itself, if possible), and
2. $w_k \notin \mathcal{FV}(\ddot{\sigma}(v'))$, for all variables $v' \in \{A, A'{:}States[A, types^A]\} \cup\ stateVars^A \cup\ postVars^A \cup\ vars^A \cup\ \mathcal{FV}(params^\pi) \cup\ \{A, A'{:}Locals[A, types^A, kind, \pi, c]\} \cup\ \{v_l, v'_l \mid l < k\}$.

Note that if $i_k = j_k$ for all k, then $\omega_{S,\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}$ is the identity operator renaming.

17.5.5 Hidden clauses

If c is a clause in a **hidden** statement

$$\pi(params^\pi)\ \textbf{where}\ p$$

where $params^\pi$ is a sequence of terms and p is a predicate, then $\ddot{\sigma}(c)$ is

$$\pi(\ddot{\sigma}_{\{v_1,v_2,\ldots\}\to\{w_1,w_2,\ldots\}}(params^\pi)\ \ldots\textbf{where}\ \ddot{\sigma}_{\{v_1,v_2,\ldots\}\to\{w_1,w_2,\ldots\}}(p)$$

where

1. v_k is a variable $i_k{:}T_k \in \mathcal{FV}(params^\pi)$
2. w_k is a variable (v_k itself, if possible) with sort T_k
3. $w_k \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(params^\pi) \cup\ \mathcal{FV}(p)\ \cup \{v_l \mid l \neq k\}$.

17.6 Notation

Except in definitions such as these, we do not employ separate notations for the extensions $\dot{\rho}$, $\ddot{\rho}$, ρ_ω, ρ_q, and $\dot{\varrho}$ of a resorting ρ. In particular, when applying a resorting ρ to an IOA automaton A, we write ρ for $\dot{\varrho}$. Similarly, we do not distinguish $\dot{\sigma}$ and $\ddot{\sigma}$ from a substitution σ and we write σ for $\ddot{\sigma}$.

Chapter 18

Conclusions

It feels terrible, it's all my own stuff and I don't want to look at it anymore.

— Frank Gehry [61]

18.1 Summary

The IOA compiler is a novel tool key to producing verified running code for distributed systems. The compiler enables programmers to transform IOA specifications of distributed algorithms into Java programs running on a collection of networked workstations. Other tools in and connected to the IOA toolkit allow programmers to formally verify the correctness of the *same* IOA specifications. The compiler overcomes the previous disconnect between correctness claims for formal specifications and actual system implementations.

We prove that under precisely stated conditions the compilation method preserves the safety properties of the program in the running system. The compiler takes an automaton as input and produces Java code as output. We prove the correctness of our compilation strategy by modeling the target Java code as another IOA automaton. That is, we treat the IOA compiler as a syntactic transformer from one kind of IOA automaton to another. We prove that the the target automaton implements the source automaton in the sense of trace inclusion.

Algorithm automata are connected to external system services by mediator automata. Proofs of correctness about IOA systems can be performed using models of these external system services that make all our assumptions about the behaviors of these systems explicit. To simplify these proofs we present simplified models of the system services that are implemented by the composition of these services with the mediator automata.

Our strategy for compilation is dependent on the ability to expand composite automata into primitive form. We have presented a series of syntactic transformations to perform this expansion. As part of this definition we have defined a desugared core IOA language and shown syntactic transformations to produce desugared automata from arbitrary primitive automata.

18.2 Assessment

In the context of the IOA toolkit project, the IOA compiler can be a key component in a software development environment. We demonstrate that it is feasible for programmers to write their specifications at a high level of abstraction, in terms of input/output automata; validate the specification using a variety of tools; refine the specification to a low-level design; and then automatically translate the design into code that runs on a collection of workstations communicating using standard networking protocols.

We are not aware of any previous system that integrates specification, light-weight validation tools, automated proof assistance, and automated code-generation. While specifications can conform closely to actual system code, a disconnect has existed in all previous systems. A human must translate the specification into an imperative language description. This disconnect between the formal specification of a distributed system and the actual implementation of that system creates several problems.

Bridging the existing disconnect between formal specification and system implementation solves several existing problems and create many new opportunities for distributed system work. At a minimum, coding the system in a programming language distinct from the specification language requires a duplication of effort. The task of the system implementor has significant overlaps with that of the specification writer. They are, after all, both striving to construct descriptions of the same system.

The separation of the specification and implementation processes allows for the introduction of transcription and interpretive errors or the failure to notice modeling errors. Transcription errors are simple copying errors that result because humans do not perform copying reliably. Interpretive errors are errors of understanding of one sort or another. The formal specification may contain ambiguous abstractions. The system implementer, less familiar with the specification language than the specification writer, may misinterpret the intent of said writer. Alternatively, the implementer may introduce "optimizations" that cause errors. (The optimizations may either be to the coding process or to the code itself.) Modeling errors occur when the specification does not adequately reflect the behavior of the real world. The overhead of the specification process may discourage the system implementer from pointing out such errors in the specification model. As a result, any confidence in the correctness of the specification transfers to the implemented system only in

substantially diminished form.

Automating the process of translating IOA specifications into running code has power beyond eliminating errors and duplicated effort. By substantially increasing the automation in the specification process, the IOA toolkit will substantially lower the barriers between formal methods and system building work. With the addition of the IOA compiler, the IOA toolkit becomes a testbed for distributed system designers. Algorithm writers will have a relatively simple path to create and measure running prototypes of high-level designs. As a result, we hope to lower the implementation expertise necessary to create even a simple prototype of a distributed system. Similarly, the formal IOA language, automated validation assistants, and clear connection to running code decrease the amount of mathematical sophistication needed to perform basic validation work. Ultimately, we hope tools like these research prototypes will lower the barriers and increase the amount of cooperation and cross-fertilization between these two aspects of the distributed systems field.

By decreasing the costs involved, the IOA toolkit should promote the iterative nature of the formal specification and system building process. Changes to the specification become easier to make. Where previously any change required tediously revisiting each step of hand proofs and hand translations, in many cases we can revalidate the design automatically by reusing automated proof scripts. This should raise the level of abstraction at which system design takes place and should result in more maintainable designs.

18.3 Future Work

While the IOA compiler and composer prototypes are fully functional, a great deal of work can be done to explore and extend the capabilities of both. Areas of inquiry for the compiler include some fixes to and extensions of the underlying IOA language, further analysis of the existing prototypes, support for a variety of network services, better handling of schedules, improved performance of datatype implementations, and direct compilation of composite automata. The composer could be extended to simplify its output and to handle nested composition.

18.3.1 IOA Language Extensions

Currently, variables in IOA can be introduced in the state of an automaton, local to transitions, or in an NDR schedule block. (Values may also be given names as automaton or action parameters.) There are a number of additional contexts in which it will be helpful to be able to declare or access variables. First, variables introduced in the schedule should be usable in the **initially det** block. In our spanning tree and broadcast case studies, we were forced to add dummy state variables to the automaton to work around this shortcoming. Second, it should be possible to declare *derived* or *history* variables outside primitive automata. In Chapter 4, we resorted to defining the auxiliary

`toSend` LSL trait because we were not able to declare variables in a simulation relation. Similarly, it would be useful to allow composite automata to declare additional history variables that are not part of the state of any one component.

A substantially more ambitious goal would be to support compilation of the extended TIOA language for timed I/O automata [66]. Among the substantial challenges in such an extension would be to include support for functions of continuous variables and real time deadlines.

18.3.2 Case studies

Further investigation of the current prototype is warranted. In particular, further studies of the effects of scaling on our implementation should be taken. Our current testbed encompasses only ten nodes due to incompatibilities among our MPI installations. At such a small scale we already observed some behavior that was difficult to understand. It is not at all clear why the LCR system exhibited bimodal runtimes. Once the cause for this behavior has been identified and fixed, scaling studies should investigate the behavior of the system on the networks orders of magnitude larger than our original testbed. We are optimistic that such studies will find that the system scales well due to the absence of global synchronization in our implementation.

Simple changes to the code such as forcing threads to sleep have already improved performance by orders of magnitude. Further measurements should be performed to help tune the system. It is currently not clear where where the system is spending most of its execution time. Identifying the bottlenecks will be the key to obtaining better performance.

18.3.3 Alternative network services

The choice to use reliable, one-way, FIFO channels as our abstract network model and MPI as our concrete channel implementation was fairly arbitrary. Expanding our model of MPI behavior and the number of MPI procedure calls supported would be one way to enhance the capabilities of the system. Alternatively, an entirely different network service could be used. Obvious possible choices include TCP, Java Remote Method Invocation (RMI), and the Java Message Service [101, 111, 116, 117]. To use a new network interconnect a designer would need to write IOA automata to model both the new concrete service being used and the new abstract channel programmers should expect. The designer would then write new mediator automata to bridge any gap between those models. A simulation proof similar to that in Chapter 4 would show that the composition of the concrete channel and mediator automata implement the abstract channel. Finally, the designer would need to alter the compiler to handle IOA actions that model calls to the network service as special cases.

It is possible one might even develop a network interface implementation registry (analogous to the current datatype implementation registry) that specifies the implementation methods for

state initialization and for pairs of network call and return actions. Such a registry would allow programmers to design their own network interconnects without substantial changes to the compiler.

Changing the network interconnect would allow programmers to explore important features of distributed systems not supported by MPI. In particular, MPI has no failure model. If one node fails, the whole system shuts down. Other network interconnects would allow the study of fault-tolerant algorithms. Similarly, MPI has a static network model. Nodes may neither join nor leave a running system. Other network models would be useful for the investigation and implementation of dynamic algorithms.

Note that abstract service correctness proofs are concerned with a higher level of abstraction than the general compiler correctness proof given in Chapter 7. The general node-by-node correctness proof is concerned only with the concrete service interface and does not assume any special properties about the transitions that implement the MPI service interface. Therefore, as long as the automaton that models the new concrete service uses pairs of input and output transitions to model method invocations and returns, Theorem 7.2 will still apply to the compiler design.

18.3.4 NDR and Schedules

For any particular algorithm, writing a better schedules may improve performance by executing fewer transitions. It would be particularly worthwhile to find the correct balance point between probing for messages too often and not probing often enough. It might be possible to develop a general characterization of such schedules.

A review of the schedules for the three case studies we performed shows many commonalities among them. For example, all three follow similar patterns when scheduling transitions derived from the mediator automata. It should be possible to schedule the mediator automata automatically. Other parts of the schedules might be generated more automatically with some minimal guidance from the programmer. For example, most **fire** statements are guarded by predicates that resemble the precondition of the transition to be executed. Identifying the correct abstraction to allow programmers to supply the minimum critical information would be key to removing much of the manual drudgery from writing schedules. Certainly the schedules for many common cases could be generated automatically.

The possible elimination of NDR annotations altogether is another alternative worth investigating. Using a scheme similar to our external representations for action invocations, each automaton could implicitly declare sorts for its action labels and parameter patterns. The automaton could then explicitly assign values to a program counter and next-parameter variable. Programs would be admissible for compilation only if next-action and next-state deterministic, as given in [120]. Alternatively, formal semantics for NDR could be given by desugaring NDR programs into such a next-action deterministic form. The result would be a new IOA automaton that implements the

original nondeterministic one.

18.3.5 Mutable datatypes

One way to improve performance would be to optimize our datatype implementations. The standard library implements every instance of a datatype as an immutable object. Even collections such as arrays, maps, sets, and sequences are implemented this way. Thus, in the current implementation every change to a variable that stores a collection entails copying all the unchanged parts of the collection to generate the new instance. Our broadcast case study exhibited the dramatic effects such inefficiencies cause when the state gets large. We chose this naïve implementation of collection datatypes for our initial implementation to avoid semantic errors than can result from shared references to mutable objects. Standard static analysis techniques should be used at compile time to detect the (usual) cases when there is no potential for shared references. In such cases, mutable collections should be used.

18.3.6 Composition

The IOA compiler could be extended to emit code for composite automata. Solovey extended IOA to support NDR schedule blocks for composite automata and the IOA simulator to simulate such automata [114]. A similar approach could work for the IOA compiler. Alternatively, each component could run in its own thread. Shared input and output actions can be compiled into a single action in the thread controlling the output action. Variables accessed by the input action would then have to be protected by locks. Such a multithreaded implementation would follow the pattern we used to implement the composition of a buffer automaton with an algorithm automaton.

A proof of correctness for such a multithreaded implementation could also follow the pattern of our proof of Theorem 7.1. Now there would be multiple macro- and micro-node automata. Since each micro-component automaton would run in a separate thread, the proof would have to be generalized to account for an arbitrary number of threads. On the other hand, because each thread would belong to a different micro-component automaton, there would be no variables (other than locks) shared across threads. Therefore, the proof of a precondition stability invariant analogous to Invariant 7.7 would not rely on assumptions about the behavior of the shared actions. (The input and output threads would still remain and would have to be treated as special cases using the logic of the proofs in Chapter 7.)

Many systems described with I/O automata are specified at several levels of abstraction. An algorithm may depend on a network service which in turn be distributed over several nodes. As a result, IOA systems may be defined as a composition of composite automata. While our definition for the expansion of composite automata into primitive form can be applied iteratively (from the inside out) in such situations, currently the composer tool does not support nested composite

automata. Implementing an iterative approach to expanding nested composite automata should be straightforward. A more direct expansion should also be investigated.

Treating the aggregate states of component automata as tuples gives rise to the "dotted form" of variable references (*e.g.*, `A.v`). Support for nested composite automata would simply increase the depth and complexity of such dotted forms (*e.g.*, `C[i].B[x,y].A.v`). In the literature, such forms are often abbreviated when no ambiguity results. For example, if `A` is the only component in the entire system to have a state variable `v` then referring to the `v` would be unambiguous. The static semantic checker should support such abbreviations.

The current definition of composition specifies `initally` **`det`** blocks as part of the expanded primitive automaton in almost all cases. Worse, these predicates contain universal quantifiers if any component is parameterized. Currently, the compiler cannot even generate runtime checks to verify that a value satisfies such a quantified predicate. Also, such universal quantifiers make any attempt to automatically generate schedules for automata very difficult. It may be possible to rework the definition of composition to eliminate such quantified terms. At the very least, it should be possible to identify classes of composite automata for which such quantified terms are either unnecessary or easy to check.

Finally, while our definition of composition in Part II is quite detailed, it remains to prove that the definition is necessarily correct. Our definition gives a syntactic manipulation of IOA programs. To prove the correctness of these manipulations one would show that the I/O automata denoted by the expanded IOA program produced by the syntactic manipulations is equivalent to the I/O automaton that results from applying the I/O automaton composition operator to the I/O automata denoted by the component automata.

Separately, proving the signature compatibility claim (Claim 15.1) and the analogous transition compatibility claims alluded to in Chapter 15 would verify that applying our syntactic manipulations to compatible automata always results in some valid primitive automaton.

Appendix A

Expanded and scheduled automata

This appendix contains the expanded primitive form of several composite automata mentioned in the text. The IOA specifications shown here were produced automatically by the composer tool from the composite node specifications given. For the three latter automata the **initially det** block and **schedule** NDR annotations were written by hand and added after composition. These schedules were used to in the measurements reported in Chapter 8.

A.1 LCRNode

```
axioms Infinite(Handle for T)

automaton LCRNode(MPIrank, MPIsize, name: Int)
  signature
    output receive(N6: Int, N7: Int)
      where N7 = MPIrank ∧ N6 = mod(MPIrank - 1, MPIsize)
    internal SEND(m: Int, N10: Int, N11: Int)
      where N11 = mod(MPIrank + 1, MPIsize) ∧ N10 = MPIrank
    input resp_Iprobe(flag: Bool, N4: Int, N5: Int)
      where N5 = MPIrank ∧ N4 = mod(MPIrank - 1, MPIsize)
    input resp_test(flag: Bool, N18: Int, N19: Int)
      where N19 = mod(MPIrank + 1, MPIsize) ∧ N18 = MPIrank
    input resp_receive(m: Int, N8: Int, N9: Int)
      where N9 = MPIrank ∧ N8 = mod(MPIrank - 1, MPIsize)
    internal vote(I0: Int) where I0 = MPIrank
    output test(handle: Handle, N16: Int, N17: Int)
      where N17 = mod(MPIrank + 1, MPIsize) ∧ N16 = MPIrank
    output Iprobe(N2: Int, N3: Int)
      where N3 = MPIrank ∧ N2 = mod(MPIrank - 1, MPIsize)
```

```
  internal RECEIVE(m: Int, I1: Int, I2: Int)
    where I2 = MPIrank ∧ I1 = mod(MPIrank - 1, MPIsize)
  internal leader(I5: Int) where I5 = MPIrank
  input resp_Isend(handle: Handle, N14: Int, N15: Int)
    where N15 = mod(MPIrank + 1, MPIsize) ∧ N14 = MPIrank
  output Isend(m: Int, N12: Int, N13: Int)
    where N13 = mod(MPIrank + 1, MPIsize) ∧ N12 = MPIrank
states
  P: _States[LCRProcess],
  RM: Map[Int, _States[ReceiveMediator, Int, Int]],
  SM: Map[Int, _States[SendMediator, Int, Int]],
  I: _States[LCRProcessInterface]
   initially
    (true ⇒ P.pending = {name} ∧ P.status = idle)
      ∧
        ∀ j: Int
           ((j = mod(MPIrank - 1, MPIsize)
               ⇒
                  RM[j].status = idle
                    ∧ RM[j].toRecv = {}
                    ∧ RM[j].ready = false)
              ∧ (j = mod(MPIrank - 1, MPIsize) ⇔ defined(RM, j)))
      ∧
        ∀ j: Int
           ((j = mod(MPIrank + 1, MPIsize)
               ⇒
                  SM[j].status = idle
                    ∧ SM[j].toSend = {}
                    ∧ SM[j].sent = {}
                    ∧ SM[j].handles = {})
              ∧ (j = mod(MPIrank + 1, MPIsize) ⇔ defined(SM, j)))
      ∧ (true ⇒ I.stdin = {} ∧ I.stdout = {})
transitions
  output receive(N6, N7)
    pre RM[mod(MPIrank - 1, MPIsize)].ready = true
          ∧ RM[mod(MPIrank - 1, MPIsize)].status = idle
    eff RM[mod(MPIrank - 1, MPIsize)].status := receive
  internal SEND(m, N10, N11)
    pre P.status ≠ idle ∧ m ∈ (P.pending)
```

```
  eff SM[mod(MPIrank + 1, MPIsize)].toSend :=
        (SM[mod(MPIrank + 1, MPIsize)].toSend) ⊢ m;
      P.pending := delete(m, P.pending)
input resp_Iprobe(flag, N4, N5)
  eff RM[mod(MPIrank - 1, MPIsize)].ready := flag;
      RM[mod(MPIrank - 1, MPIsize)].status := idle
input resp_test(flag, N18, N19)
  eff if flag then
        SM[mod(MPIrank + 1, MPIsize)].handles :=
          tail(SM[mod(MPIrank + 1, MPIsize)].handles);
        SM[mod(MPIrank + 1, MPIsize)].sent :=
          tail(SM[mod(MPIrank + 1, MPIsize)].sent)
      fi;
      SM[mod(MPIrank + 1, MPIsize)].status := idle
input resp_receive(m, N8, N9)
  eff RM[mod(MPIrank - 1, MPIsize)].toRecv :=
        (RM[mod(MPIrank - 1, MPIsize)].toRecv) ⊢ m;
      RM[mod(MPIrank - 1, MPIsize)].ready := false;
      RM[mod(MPIrank - 1, MPIsize)].status := idle
internal vote(I0)
  pre head(I.stdin).action = vote
        ∧ len(head(I.stdin).params) = 1
        ∧ tag(head(I.stdin).params[0]) = Int
        ∧ head(I.stdin).params[0].Int = I0
  eff P.status := voting;
      I.stdin := tail(I.stdin)
output test(handle, N16, N17)
  pre SM[mod(MPIrank + 1, MPIsize)].status = idle
        ∧ handle = head(SM[mod(MPIrank + 1, MPIsize)].handles)
  eff SM[mod(MPIrank + 1, MPIsize)].status := test
output Iprobe(N2, N3)
  pre RM[mod(MPIrank - 1, MPIsize)].status = idle
        ∧ RM[mod(MPIrank - 1, MPIsize)].ready = false
  eff RM[mod(MPIrank - 1, MPIsize)].status := Iprobe
internal RECEIVE(m, I1, I2) where m > name ∨ m < name ∨ m = name
  pre m = head(RM[mod(MPIrank - 1, MPIsize)].toRecv)
  eff if m > name then P.pending := insert(m, P.pending)
      elseif m = name then P.status := elected
      fi;
```

```
        RM[mod(MPIrank - 1, MPIsize)].toRecv :=
          tail(RM[mod(MPIrank - 1, MPIsize)].toRecv)
    internal leader(I5)
      pre P.status = elected
      eff I.stdout := (I.stdout) ⊢ [leader, {} ⊢ Int(I5)];
          P.status := announced
    input resp_Isend(handle, N14, N15)
      eff SM[mod(MPIrank + 1, MPIsize)].handles :=
            (SM[mod(MPIrank + 1, MPIsize)].handles) ⊢ handle;
          SM[mod(MPIrank + 1, MPIsize)].status := idle
    output Isend(m, N12, N13)
      pre head(SM[mod(MPIrank + 1, MPIsize)].toSend) = m
            ∧ SM[mod(MPIrank + 1, MPIsize)].status = idle
      eff SM[mod(MPIrank + 1, MPIsize)].toSend :=
            tail(SM[mod(MPIrank + 1, MPIsize)].toSend);
          SM[mod(MPIrank + 1, MPIsize)].sent :=
            (SM[mod(MPIrank + 1, MPIsize)].sent) ⊢ m;
          SM[mod(MPIrank + 1, MPIsize)].status := Isend


type Status = enumeration of idle, voting, elected, announced

type rCall = enumeration of idle, receive, Iprobe

type sCall = enumeration of idle, Isend, test

type Status = enumeration of idle, voting, elected, announced

type IOA_Invocation = tuple of action: IOA_Action, params:
  Seq[IOA_Parameter]

type IOA_Parameter = union of Int: Int

type IOA_Action = enumeration of RECEIVE, SEND, leader, vote

type _States[LCRProcess] = tuple of pending: Mset[Int], status: Status

type _States[ReceiveMediator, Int, Int] = tuple of status: rCall, toRecv:
  Seq[Int], ready: Bool
```

```
type _States[SendMediator, Int, Int] = tuple of status: sCall, toSend:
  Seq[Int], sent: Seq[Int], handles: Seq[Handle]

type _States[LCRProcessInterface] = tuple of stdin: LSeqIn[IOA_Invocation],
  stdout: LSeqOut[IOA_Invocation]
```

A.2 TerminatingLCR

```
axioms Infinite(Handle for T)
axioms ChoiceMset(Int)

automaton TerminatingLCRNode(MPIrank, MPIsize, name: Int)
  signature
    output receive(N6: Int, N7: Int)
      where N7 = MPIrank ∧ N6 = mod(MPIrank - 1, MPIsize)
    internal SEND(m: Int, I3: Int, I4: Int)
      where I4 = mod(MPIrank + 1, MPIsize) ∧ I3 = MPIrank
    input resp_Iprobe(flag: Bool, N4: Int, N5: Int)
      where N5 = MPIrank ∧ N4 = mod(MPIrank - 1, MPIsize)
    input resp_test(flag: Bool, N18: Int, N19: Int)
      where N19 = mod(MPIrank + 1, MPIsize) ∧ N18 = MPIrank
    input resp_receive(m: Int, N8: Int, N9: Int)
      where N9 = MPIrank ∧ N8 = mod(MPIrank - 1, MPIsize)
    internal vote(I0: Int) where I0 = MPIrank
    output test(handle: Handle, N16: Int, N17: Int)
      where N17 = mod(MPIrank + 1, MPIsize) ∧ N16 = MPIrank
    output Iprobe(N2: Int, N3: Int)
      where N3 = MPIrank ∧ N2 = mod(MPIrank - 1, MPIsize)
    internal RECEIVE(m: Int, I1: Int, I2: Int)
      where I2 = MPIrank ∧ I1 = mod(MPIrank - 1, MPIsize)
    internal leader(I5: Int) where I5 = MPIrank
    input resp_Isend(handle: Handle, N14: Int, N15: Int)
      where N15 = mod(MPIrank + 1, MPIsize) ∧ N14 = MPIrank
    output Isend(m: Int, N12: Int, N13: Int)
      where N13 = mod(MPIrank + 1, MPIsize) ∧ N12 = MPIrank
  states
    P: _States[LCRProcess],
    RM: Map[Int, _States[ReceiveMediator, Int, Int]],
    SM: Map[Int, _States[SendMediator, Int, Int]],
    I: _States[LCRProcessInterface]
     initially
      (true ⇒ P.pending = {name} ∧ P.status = idle)
        ∧
          ∀ j: Int
             ((j = mod(MPIrank - 1, MPIsize)
                 ⇒
```

```
                RM[j].status = idle
                  ∧ RM[j].toRecv = {}
                  ∧ RM[j].ready = false)
              ∧ (j = mod(MPIrank - 1, MPIsize) ⇔ defined(RM, j)))
      ∧
        ∀ j: Int
           ((j = mod(MPIrank + 1, MPIsize)
              ⇒
                SM[j].status = idle
                  ∧ SM[j].toSend = {}
                  ∧ SM[j].sent = {}
                  ∧ SM[j].handles = {})
              ∧ (j = mod(MPIrank + 1, MPIsize) ⇔ defined(SM, j)))
        ∧ (true ⇒ I.stdin = {} ∧ I.stdout = {})
    det do
          P := [{name}, idle];
          RM := update(empty, mod(MPIrank-1,MPIsize), [idle, {}, false]);
          SM := update(empty, mod(MPIrank+1,MPIsize), [idle, {}, {}, {}]);
          I := [{}, {}]
        od
transitions
  output receive(N6, N7)
    pre RM[mod(MPIrank - 1, MPIsize)].ready = true
          ∧ RM[mod(MPIrank - 1, MPIsize)].status = idle
    eff RM[mod(MPIrank - 1, MPIsize)].status := receive
  internal SEND(m, I3, I4)
    pre P.status ≠ idle ∧ m ∈ (P.pending)
    eff SM[mod(MPIrank + 1, MPIsize)].toSend :=
          (SM[mod(MPIrank + 1, MPIsize)].toSend) ⊢ m;
        P.pending := delete(m, P.pending)
  input resp_Iprobe(flag, N4, N5)
    eff RM[mod(MPIrank - 1, MPIsize)].ready := flag;
        RM[mod(MPIrank - 1, MPIsize)].status := idle
  input resp_test(flag, N18, N19)
    eff if flag then
          SM[mod(MPIrank + 1, MPIsize)].handles :=
            tail(SM[mod(MPIrank + 1, MPIsize)].handles);
          SM[mod(MPIrank + 1, MPIsize)].sent :=
            tail(SM[mod(MPIrank + 1, MPIsize)].sent)
```

```
      fi;
      SM[mod(MPIrank + 1, MPIsize)].status := idle
input resp_receive(m, N8, N9)
  eff RM[mod(MPIrank - 1, MPIsize)].toRecv :=
        (RM[mod(MPIrank - 1, MPIsize)].toRecv) ⊢ m;
      RM[mod(MPIrank - 1, MPIsize)].ready := false;
      RM[mod(MPIrank - 1, MPIsize)].status := idle
internal vote(I0)
  pre head(I.stdin).action = vote
        ∧ len(head(I.stdin).params) = 1
        ∧ tag(head(I.stdin).params[0]) = Int
        ∧ head(I.stdin).params[0].Int = I0
  eff P.status := voting;
      I.stdin := tail(I.stdin)
output test(handle, N16, N17)
  pre SM[mod(MPIrank + 1, MPIsize)].status = idle
        ∧ handle = head(SM[mod(MPIrank + 1, MPIsize)].handles)
  eff SM[mod(MPIrank + 1, MPIsize)].status := test
output Iprobe(N2, N3)
  pre RM[mod(MPIrank - 1, MPIsize)].status = idle
        ∧ RM[mod(MPIrank - 1, MPIsize)].ready = false
  eff RM[mod(MPIrank - 1, MPIsize)].status := Iprobe
internal
  RECEIVE(m, I1, I2)
    where m > name ∨ (0 ≤ m ∧ m < name) ∨ m < 0 ∨ m = name
  pre m = head(RM[mod(MPIrank - 1, MPIsize)].toRecv)
  eff if m > name then P.pending := insert(m, P.pending)
      elseif m < 0 then
         if P.status ≠ announced then
            P.pending := insert(m, P.pending)
         fi;
         P.status := over
      elseif m = name then P.status := elected
      fi;
      RM[mod(MPIrank - 1, MPIsize)].toRecv :=
        tail(RM[mod(MPIrank - 1, MPIsize)].toRecv)
internal leader(I5)
  pre P.status = elected
  eff I.stdout := (I.stdout) ⊢ [leader, {} ⊢ Int(I5)];
```

```
        P.status := announced;
        P.pending := insert(-1, P.pending)
    input resp_Isend(handle, N14, N15)
      eff SM[mod(MPIrank + 1, MPIsize)].handles :=
            (SM[mod(MPIrank + 1, MPIsize)].handles) ⊢ handle;
        SM[mod(MPIrank + 1, MPIsize)].status := idle
    output Isend(m, N12, N13)
      pre head(SM[mod(MPIrank + 1, MPIsize)].toSend) = m
          ∧ SM[mod(MPIrank + 1, MPIsize)].status = idle
      eff SM[mod(MPIrank + 1, MPIsize)].toSend :=
            tail(SM[mod(MPIrank + 1, MPIsize)].toSend);
        SM[mod(MPIrank + 1, MPIsize)].sent :=
          (SM[mod(MPIrank + 1, MPIsize)].sent) ⊢ m;
        SM[mod(MPIrank + 1, MPIsize)].status := Isend
schedule
do
  while ¬(P.status = over ∧
        size(P.pending) = 0 ∧
        SM[mod(MPIrank + 1, MPIsize)].toSend = {} ∧
        SM[mod(MPIrank + 1, MPIsize)].handles = {}) do
    if P.status = elected then fire internal leader(MPIrank) fi;
    if len(I.stdin) > 0
       ∧ head(I.stdin).action = vote
       ∧ len(head(I.stdin).params) = 1
       ∧ tag(head(I.stdin).params[0]) = Int
       ∧ head(I.stdin).params[0].Int = MPIrank then
       fire internal vote(MPIrank)
    fi;
    if P.status ≠ idle ∧ size(P.pending) ≠ 0 then
       fire internal SEND(chooseRandom(P.pending), MPIrank,
                          mod(MPIrank + 1, MPIsize))
    fi;
    if SM[mod(MPIrank + 1, MPIsize)].status = idle ∧
       SM[mod(MPIrank + 1, MPIsize)].toSend ≠ {} then
       fire output Isend(head(SM[mod(MPIrank + 1, MPIsize)].toSend),
                         MPIrank, mod(MPIrank + 1, MPIsize))
    fi;
    if SM[mod(MPIrank + 1, MPIsize)].status = idle ∧
       SM[mod(MPIrank + 1, MPIsize)].handles ≠ {} then
```

```
      fire output test(head(SM[mod(MPIrank + 1, MPIsize)].handles),
                       MPIrank, mod(MPIrank + 1, MPIsize))
    fi;
    if RM[mod(MPIrank - 1, MPIsize)].status = idle ∧
       RM[mod(MPIrank - 1, MPIsize)].ready = false then
       fire output Iprobe(MPIrank, mod(MPIrank - 1, MPIsize))
    fi;
    if RM[mod(MPIrank - 1, MPIsize)].status = idle ∧
       RM[mod(MPIrank - 1, MPIsize)].ready = true then
       fire output receive(MPIrank, mod(MPIrank - 1, MPIsize))
    fi;
    if RM[mod(MPIrank - 1, MPIsize)].toRecv ≠ {} then
       fire internal RECEIVE(head(RM[mod(MPIrank - 1, MPIsize)].toRecv),
                             mod(MPIrank - 1, MPIsize), MPIrank)
    fi
  od
od


type Status = enumeration of idle, voting, elected, announced, over

type rCall = enumeration of idle, receive, Iprobe

type sCall = enumeration of idle, Isend, test

type Status = enumeration of idle, voting, elected, announced, over

type IOA_Invocation = tuple of action: IOA_Action, params:
  Seq[IOA_Parameter]

type IOA_Parameter = union of Int: Int

type IOA_Action = enumeration of RECEIVE, SEND, leader, vote

type _States[LCRProcess] = tuple of pending: Mset[Int], status: Status

type _States[ReceiveMediator, Int, Int] = tuple of status: rCall, toRecv:
  Seq[Int], ready: Bool
```

```
type _States[SendMediator, Int, Int] = tuple of status: sCall, toSend:
  Seq[Int], sent: Seq[Int], handles: Seq[Handle]

type _States[LCRProcessInterface] = tuple of stdin: LSeqIn[IOA_Invocation],
  stdout: LSeqOut[IOA_Invocation]
```

A.3 spanNode

```
axioms Infinite(Handle for T)

axioms ChoiceSet(Int for E)

automaton spanNode(MPIrank, MPIsize: Int, neighbors: Set[Int])
  signature
    output receive(N6: Int, N7: Int) where N6 = MPIrank
    internal SEND(m: Message, N10: Int, N11: Int) where N10 = MPIrank
    input resp_Iprobe(flag: Bool, N4: Int, N5: Int) where N4 = MPIrank
    input resp_test(flag: Bool, N18: Int, N19: Int) where N18 = MPIrank
    input resp_receive(m: Message, N8: Int, N9: Int) where N8 = MPIrank
    output test(handle: Handle, N16: Int, N17: Int) where N16 = MPIrank
    output Iprobe(N2: Int, N3: Int) where N2 = MPIrank
    internal RECEIVE(m: Message, N0: Int, N1: Int) where N0 = MPIrank
    input resp_Isend(handle: Handle, N14: Int, N15: Int)
      where N14 = MPIrank
    internal parent(I3: Int, j6: Int) where I3 = MPIrank
    output Isend(m: Message, N12: Int, N13: Int) where N12 = MPIrank
    internal search(I0: Int) where I0 = MPIrank
  states
    P: _States[spanProcess],
    RM: Map[Int, _States[ReceiveMediator, Message, Int]],
    SM: Map[Int, _States[SendMediator, Message, Int]],
    I: _States[spanProcessInterface]
     initially
      (true
         ⇒
           ∀ k: Int (defined(P.send, k) ⇔ k ∈ neighbors)
             ∧
               ∀ k: Int
                  (defined(P.send, k) ∧ P.send[k] = search
                     ⇔ MPIrank = 0)
             ∧ P.searching = false
             ∧ P.reported = false
             ∧ P.parent = -1)
        ∧
          ∀ j: Int
             (RM[j].status = idle
```

```
            ∧ RM[j].toRecv = {}
            ∧ RM[j].ready = false
            ∧ defined(RM, j))
    ∧
      ∀ j: Int
         (SM[j].status = idle
            ∧ SM[j].toSend = {}
            ∧ SM[j].sent = {}
            ∧ SM[j].handles = {}
            ∧ defined(SM, j))
    ∧ (true ⇒ I.stdin = {} ∧ I.stdout = {})
  det
    do
      I := [{}, {}];
      P := [false, false, -1, empty, neighbors, -1];
      RM := empty;
      SM := empty;
      while ¬isEmpty(P.nbrs) do
        P.nbr := chooseRandom(P.nbrs);
        P.nbrs := rest(P.nbrs);
        if 0 ≤ P.nbr ∧ P.nbr < MPIsize then % Sanity check neighbors set
          P.send[P.nbr] := if MPIrank = 0 then search else nil;
          RM[P.nbr] := [idle, {}, false];
          SM[P.nbr] := [idle, {}, {}, {}]
        fi
      od
    od

transitions
  output receive(N6, N7)
    pre RM[N7].ready = true ∧ RM[N7].status = idle
    eff RM[N7].status := receive
  internal SEND(m, N10, N11) where N10 = MPIrank
    pre P.searching ∧ P.send[N11] = search ∧ m = search
    eff SM[N11].toSend := (SM[N11].toSend) ⊢ m;
        P.send[N11] := nil
  input resp_Iprobe(flag, N4, N5)
    eff RM[N5].ready := flag;
        RM[N5].status := idle
```

```
input resp_test(flag, N18, N19)
  eff if flag then
        SM[N19].handles := tail(SM[N19].handles);
        SM[N19].sent := tail(SM[N19].sent)
      fi;
      SM[N19].status := idle
input resp_receive(m, N8, N9)
  eff RM[N9].toRecv := (RM[N9].toRecv) ⊢ m;
      RM[N9].ready := false;
      RM[N9].status := idle
output test(handle, N16, N17)
  pre SM[N17].status = idle ∧ handle = head(SM[N17].handles)
  eff SM[N17].status := test
output Iprobe(N2, N3)
  pre RM[N3].status = idle ∧ RM[N3].ready = false
  eff RM[N3].status := Iprobe
internal RECEIVE(m, N0, N1) where MPIrank = 0 ∨ MPIrank ≠ 0
  pre m = head(RM[N1].toRecv)
  eff if MPIrank ≠ 0 then
        if P.parent = -1 then
          P.parent := N1;
          for k: Int in neighbors - {N1} do
            P.send[k] := search
            od
        fi
      fi;
      RM[N1].toRecv := tail(RM[N1].toRecv)
input resp_Isend(handle, N14, N15)
  eff SM[N15].handles := (SM[N15].handles) ⊢ handle;
      SM[N15].status := idle
internal parent(I3, j6)
  pre (P.parent = j6
        ∧ (P.parent) ≥ 0
        ∧ ¬(P.reported)
        ∧ P.searching
        ∧ MPIrank ≠ 0)
       ∨ (¬(P.reported) ∧ P.searching ∧ MPIrank = 0)
  eff I.stdout := (I.stdout) ⊢ [parent, {} ⊢ Int(I3) ⊢ Int(j6)];
      if MPIrank ≠ 0 then P.reported := true
```

```
        elseif MPIrank = 0 then P.reported := true
        fi
    output Isend(m, N12, N13)
      pre head(SM[N13].toSend) = m ∧ SM[N13].status = idle
      eff SM[N13].toSend := tail(SM[N13].toSend);
        SM[N13].sent := (SM[N13].sent) ⊢ m;
        SM[N13].status := Isend
    internal search(I0)
      pre head(I.stdin).action = search
          ∧ len(head(I.stdin).params) = 1
          ∧ tag(head(I.stdin).params[0]) = Int
          ∧ head(I.stdin).params[0].Int = I0
      eff P.searching := true;
        I.stdin := tail(I.stdin)

schedule
states
  ngbrs : Set[Int],
  ngbr: Int,
  flooding: Bool := true
do
  while ¬P.reported ∨ flooding do
    if P.searching ∧ ¬P.reported ∧ (P.parent ≥ 0 ∨ MPIrank = 0) then
       fire internal parent(MPIrank, P.parent)
    fi;
    if len(I.stdin) > 0
       ∧ head(I.stdin).action = search
       ∧ len(head(I.stdin).params) = 1
       ∧ tag(head(I.stdin).params[0]) = Int then
       ∧ head(I.stdin).params[0].Int = MPIrank then
       fire internal search(MPIrank)
    fi;
    ngbrs := neighbors;
    while ¬isEmpty(ngbrs) do
      ngbr := chooseRandom(ngbrs);
      ngbrs := rest(ngbrs);
      if 0 ≤ ngbr ∧ ngbr < MPIsize then  % Sanity check neighbors set
         if P.searching ∧ P.send[ngbr] = search then
            fire internal SEND(search, MPIrank, ngbr)
```

```
          fi;
          if SM[ngbr].status = idle ∧ SM[ngbr].toSend ≠ {} then
             fire output Isend(head(SM[ngbr].toSend), MPIrank, ngbr)
          fi;
          if SM[ngbr].status = idle ∧ SM[ngbr].handles ≠ {} then
             fire output test(head(SM[ngbr].handles), MPIrank, ngbr)
          fi;
          if RM[ngbr].status = idle ∧ RM[ngbr].ready = false then
            fire output Iprobe(MPIrank, ngbr)
          fi;
          if RM[ngbr].status = idle ∧  RM[ngbr].ready = true then
             fire output receive(MPIrank, ngbr)
          fi;
          if RM[ngbr].toRecv ≠ {} then
             fire internal RECEIVE(head(RM[ngbr].toRecv), MPIrank, ngbr)
          fi
        fi
      od;
      ngbrs := neighbors;
      flooding := false;
      while ¬isEmpty(ngbrs) do
        ngbr := chooseRandom(ngbrs);
        ngbrs := rest(ngbrs);
        if 0 ≤ ngbr ∧ ngbr < MPIsize then % Sanity check neighbors set
           flooding := flooding ∨ P.send[ngbr] = search ∨
                       SM[ngbr].toSend ≠ {} ∨ SM[ngbr].handles ≠ {}
        fi
      od
    od
od

type Message = enumeration of search, nil

type rCall = enumeration of idle, receive, Iprobe

type sCall = enumeration of idle, Isend, test

type Message = enumeration of search, nil
```

```
type IOA_Invocation = tuple of action: IOA_Action, params:
  Seq[IOA_Parameter]

type IOA_Parameter = union of Int: Int, Message: Message

type IOA_Action = enumeration of RECEIVE, SEND, parent, search

type _States[spanProcess] = tuple of searching: Bool, reported: Bool,
  parent: Int, send: Map[Int, Message], nbrs: Set[Int], nbr: Int

type _States[ReceiveMediator, Message, Int] = tuple of status: rCall,
  toRecv: Seq[Message], ready: Bool

type _States[SendMediator, Message, Int] = tuple of status: sCall, toSend:
  Seq[Message], sent: Seq[Message], handles: Seq[Handle]

type _States[spanProcessInterface] = tuple of stdin:
  LSeqIn[IOA_Invocation], stdout: LSeqOut[IOA_Invocation]
```

A.4 bcastNode

```
axioms Infinite(Handle for T)

axioms ChoiceSet(Int for E)

automaton bcastNode(MPIrank, MPIsize: Int, neighbors: Set[Int], last: Int)
  signature
    output receive(N6: Int, N7: Int) where N6 = MPIrank
    internal SEND(m: Message, N10: Int, N11: Int) where N10 = MPIrank
    input resp_Iprobe(flag: Bool, N4: Int, N5: Int) where N4 = MPIrank
    internal report(I7: Int, v: Int) where I7 = MPIrank
    input resp_test(flag: Bool, N18: Int, N19: Int) where N18 = MPIrank
    input resp_receive(m: Message, N8: Int, N9: Int) where N8 = MPIrank
    internal queue(I1: Int, v: Int) where I1 = MPIrank
    output test(handle: Handle, N16: Int, N17: Int) where N16 = MPIrank
    output Iprobe(N2: Int, N3: Int) where N2 = MPIrank
    internal RECEIVE(m: Message, N0: Int, N1: Int) where N0 = MPIrank
    internal bcast(I0: Int, v: Int) where I0 = MPIrank
    input resp_Isend(handle: Handle, N14: Int, N15: Int)
      where N14 = MPIrank
    internal parent(I6: Int, j: Null[Int]) where I6 = MPIrank
    internal ackLast(I3: Int) where I3 = MPIrank
    output Isend(m: Message, N12: Int, N13: Int) where N12 = MPIrank
    internal ackInit(I2: Int) where I2 = MPIrank
  states
    P: _States[bcastProcess],
    RM: Map[Int, _States[ReceiveMediator, Message, Int]],
    SM: Map[Int, _States[SendMediator, Message, Int]],
    I: _States[bcastProcessInterface]
     initially
      (true
         ⇒
           ∀ k: Int (defined(P.send, k) ⇔ k ∈ neighbors)
             ∧
               ∀ k: Int
                  (MPIrank = 0
                     ⇒
                        defined(P.send, k)
                          ∧ head(P.send[k]) = [bcast, nil])
```

```
            ∧ P.status = idle
            ∧ P.parent = nil
            ∧ P.children = {}
            ∧ P.acked = {}
            ∧ P.outgoing = {}
            ∧ P.incomming = {})
      ∧
        ∀ j: Int
            (RM[j].status = idle
               ∧ RM[j].toRecv = {}
               ∧ RM[j].ready = false
               ∧ defined(RM, j))
      ∧
        ∀ j: Int
            (SM[j].status = idle
               ∧ SM[j].toSend = {}
               ∧ SM[j].sent = {}
               ∧ SM[j].handles = {}
               ∧ defined(SM, j))
      ∧ (true ⇒ I.stdin = {} ∧ I.stdout = {})
   det
     do
        I := [{}, {}];
        P := [idle, nil, {}, {}, {}, {}, empty, neighbors, -1];
        RM := empty;
        SM := empty;
        while ¬isEmpty(P.nbrs) do
          P.nbr := chooseRandom(P.nbrs);
          P.nbrs := rest(P.nbrs);
          if 0 ≤ P.nbr ∧ P.nbr < MPIsize then % Sanity check neighbors set
             P.send[P.nbr] := {};
             RM[P.nbr] := [idle, {}, false];
             SM[P.nbr] := [idle, {}, {}, {}]
          fi
        od
     od

transitions
  output receive(N6, N7)
```

```
  pre RM[N7].ready = true ∧ RM[N7].status = idle
  eff RM[N7].status := receive
internal SEND(m, N10, N11) where N10 = MPIrank
  pre P.send[N11] ≠ {} ∧ m = head(P.send[N11])
  eff SM[N11].toSend := (SM[N11].toSend) ⊢ m;
      P.send[N11] := tail(P.send[N11])
input resp_Iprobe(flag, N4, N5)
  eff RM[N5].ready := flag;
      RM[N5].status := idle
internal report(I7, v)
  pre P.incomming ≠ {} ∧ v = head(P.incomming)
  eff I.stdout := (I.stdout) ⊢ [report, {} ⊢ Int(I7) ⊢ Int(v)];
      P.incomming := tail(P.incomming)
input resp_test(flag, N18, N19)
  eff if flag then
        SM[N19].handles := tail(SM[N19].handles);
        SM[N19].sent := tail(SM[N19].sent)
      fi;
      SM[N19].status := idle
input resp_receive(m, N8, N9)
  eff RM[N9].toRecv := (RM[N9].toRecv) ⊢ m;
      RM[N9].ready := false;
      RM[N9].status := idle
internal queue(I1, v)
  eff if MPIrank = 0 then
        P.outgoing := tail(P.outgoing);
        for k: Int in P.children do
          P.send[k] := P.send[k] ⊢ [bcast, embed(v)]
          od;
        if v = last then
          P.status := finalizing;
          P.acked := {}
        fi
      fi
output test(handle, N16, N17)
  pre SM[N17].status = idle ∧ handle = head(SM[N17].handles)
  eff SM[N17].status := test
output Iprobe(N2, N3)
  pre RM[N3].status = idle ∧ RM[N3].ready = false
```

```
  eff RM[N3].status := Iprobe
internal
  RECEIVE(m, N0, N1)
    where m.kind = nack
            ∨ m.kind = ack
            ∨ (m.kind = bcast ∧ m.payload = nil)
            ∨ (m.kind = bcast ∧ m.payload ≠ nil)
  pre m = head(RM[N1].toRecv)
  eff if m.kind = nack then P.acked := (P.acked) ∪ {N1}
      elseif m.kind = ack then
         P.acked := (P.acked) ∪ {N1};
         if P.status = initializing then
            P.children := (P.children) ∪ {N1}
         fi
      elseif m.kind = bcast ∧ m.payload = nil then
         if P.parent = nil ∧ MPIrank ≠ 0 then
            P.parent := embed(N1);
            P.status := initializing;
            P.acked := {(P.parent).val};
            for k: Int in neighbors - {(P.parent).val} do
              P.send[k] := P.send[k] ⊢ m
              od
         else P.send[N1] := P.send[N1] ⊢ [nack, nil]
         fi
      elseif m.kind = bcast ∧ m.payload ≠ nil then
         P.incomming := (P.incomming) ⊢ ((m.payload).val);
         for k: Int in P.children do
           P.send[k] := P.send[k] ⊢ m
           od;
         if (m.payload).val = last then
            P.status := finalizing;
            P.acked := {}
         fi
      fi;
      RM[N1].toRecv := tail(RM[N1].toRecv)
internal bcast(I0, v)
  pre head(I.stdin).action = bcast
        ∧ len(head(I.stdin).params) = 2
        ∧ tag(head(I.stdin).params[0]) = Int
```

```
          ∧ head(I.stdin).params[0].Int = I0
          ∧ tag(head(I.stdin).params[1]) = Int
          ∧ head(I.stdin).params[1].Int = v
    eff if MPIrank = 0 then
          if P.status = idle then
            P.status := initializing;
            for k: Int in neighbors do
              P.send[k] := P.send[k] ⊢ [bcast, nil]
              od
          fi;
          P.outgoing := (P.outgoing) ⊢ v
        fi;
        I.stdin := tail(I.stdin)
  input resp_Isend(handle, N14, N15)
    eff SM[N15].handles := (SM[N15].handles) ⊢ handle;
        SM[N15].status := idle
  internal parent(I6, j)
    pre P.status = initializing ∧ P.acked = neighbors ∧ P.parent = j
    eff I.stdout :=
          (I.stdout) ⊢ [parent, {} ⊢ Int(I6) ⊢ Null'_Int_'(j)];
        P.status := announced
  internal ackLast(I3)
    eff P.status := done;
        if MPIrank ≠ 0 then
          P.send[(P.parent).val] := P.send[(P.parent).val] ⊢ [ack, nil]
        fi
  output Isend(m, N12, N13)
    pre head(SM[N13].toSend) = m ∧ SM[N13].status = idle
    eff SM[N13].toSend := tail(SM[N13].toSend);
        SM[N13].sent := (SM[N13].sent) ⊢ m;
        SM[N13].status := Isend
  internal ackInit(I2)
    eff P.status := bcasting;
        if MPIrank ≠ 0 then
          P.send[(P.parent).val] := P.send[(P.parent).val] ⊢ [ack, nil]
        fi

schedule
states
```

```
  ngbrs : Set[Int],
  ngbr: Int,
  sending: Bool := true
do
  while P.status ≠ done ∨ sending do
    if P.status = initializing ∧ P.acked = neighbors then
       fire internal parent(MPIrank, P.parent)
    fi;
    if P.incomming ≠ {} then
       fire internal report(MPIrank, head(P.incomming))
    fi;
    if len(I.stdin) > 0
       ∧ head(I.stdin).action = bcast
       ∧ len(head(I.stdin).params) = 2
       ∧ tag(head(I.stdin).params[0]) = Int
       ∧ head(I.stdin).params[0].Int = MPIrank
       ∧ tag(head(I.stdin).params[1]) = Int then
       fire internal bcast(MPIrank, head(I.stdin).params[1].Int)
    fi;
    if P.status = announced then
       fire internal ackInit(MPIrank)
    fi;
    if P.status = finalizing ∧ P.acked = P.children then
       fire internal ackLast(MPIrank)
    fi;
    ngbrs := neighbors;
    while ¬isEmpty(ngbrs) do
      ngbr := chooseRandom(ngbrs);
      ngbrs := rest(ngbrs);
      if 0 ≤ ngbr ∧ ngbr < MPIsize then  % Sanity check neighbors set
         if P.status = bcasting ∧ P.outgoing ≠ {} then
            fire internal queue(MPIrank, head(P.outgoing))
         fi;
         if P.send[ngbr] ≠ {} then
            fire internal SEND(head(P.send[ngbr]), MPIrank, ngbr)
         fi;
         if SM[ngbr].status = idle ∧ SM[ngbr].toSend ≠ {} then
            fire output Isend(head(SM[ngbr].toSend), MPIrank, ngbr)
         fi;
```

```
          if SM[ngbr].status = idle ∧ SM[ngbr].handles ≠ {} then
             fire output test(head(SM[ngbr].handles), MPIrank, ngbr)
          fi;
          if RM[ngbr].status = idle ∧ RM[ngbr].ready = false then
            fire output Iprobe(MPIrank, ngbr)
          fi;
          if RM[ngbr].status = idle ∧  RM[ngbr].ready = true then
             fire output receive(MPIrank, ngbr)
          fi;
          if RM[ngbr].toRecv ≠ {} then
             fire internal RECEIVE(head(RM[ngbr].toRecv), MPIrank, ngbr)
          fi
        fi
      od;
      ngbrs := neighbors;
      sending := false;
      while ¬isEmpty(ngbrs) do
        ngbr := chooseRandom(ngbrs);
        ngbrs := rest(ngbrs);
        if 0 ≤ ngbr ∧ ngbr < MPIsize then % Sanity check neighbors set
           sending := sending ∨ P.send[ngbr] ≠ {} ∨
                      SM[ngbr].toSend ≠ {} ∨ SM[ngbr].handles ≠ {}
        fi
      od
    od
od



type Status = enumeration of idle, initializing, announced, bcasting,
  finalizing, done

type Kind = enumeration of bcast, ack, nack

type Message = tuple of kind: Kind, payload: Null[Int]

type rCall = enumeration of idle, receive, Iprobe

type sCall = enumeration of idle, Isend, test
```

```
type Message = tuple of kind: Kind, payload: Null[Int]

type Kind = enumeration of bcast, ack, nack

type Status = enumeration of idle, initializing, announced, bcasting,
  finalizing, done

type IOA_Invocation = tuple of action: IOA_Action, params:
  Seq[IOA_Parameter]

type IOA_Parameter = union of Int: Int, Message: Message, Null'_Int_': 
  Null[Int]

type IOA_Action = enumeration of RECEIVE, SEND, bcast, parent, report

type _States[bcastProcess] = tuple of status: Status, parent: Null[Int],
  children: Set[Int], acked: Set[Int], outgoing: Seq[Int], incomming:
  Seq[Int], send: Map[Int, Seq[Message]], nbrs: Set[Int], nbr: Int

type _States[ReceiveMediator, Message, Int] = tuple of status: rCall,
  toRecv: Seq[Message], ready: Bool

type _States[SendMediator, Message, Int] = tuple of status: sCall, toSend:
  Seq[Message], sent: Seq[Message], handles: Seq[Handle]

type _States[bcastProcessInterface] = tuple of stdin:
  LSeqIn[IOA_Invocation], stdout: LSeqOut[IOA_Invocation]
```

Bibliography

[1] INMOS Ltd: occam Programming Manual, 1984.

[2] INMOS Ltd: occam 2 Reference Manual, 1988.

[3] Myla M. Archer, Constance L. Heitmeyer, and Elvinia Riccobene. Using TAME to prove invariants of automata models: Two case studies. In *Proceedings of Third ACM Workshop on Formal Methods in Software Practice (FMSP'00)*, August 2000.

[4] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. Submitted to First UK Workshop on Java for High Performance Network Computing, Europar 1998.

[5] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[6] A. Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen*. PhD thesis, Universität Karlsruhe, 1997.

[7] A. Biere. μcke — efficient μ-calculus model checking. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 468–471. Springer Verlag, 1997.

[8] Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomás Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, July 1996.

[9] Nikolaj Bjørner, Anca Browne, Michael A. Colón, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomás Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, pages 1–45, 1999.

[10] Nikolaj Bjørner, Zohar Manna, Henny Sipma, and Tomás Uribe. Deductive verification of real-time systems using STeP. Technical Report STAN-CS-TR-98-1616, Computer Science Department, Stanford University, December 1998.

[11] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2001.

[12] Andrej Bogdanov, Stephen Garland, and Nancy Lynch. Mechanical translation of I/O automaton specifications into first-order logic. In Doron Peled and Moshe Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems (22nd International Conference on Formal Techniques for Networked and Distributed Systems - FORTE 2002, Houston, Texas, November 2002)*, volume 2529 of *Lecture Notes in Computer Science*, pages 364–368. Springer, 2002.

[13] Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14:127–146, 2001. Also, Technical Memo MIT/LCS/TM-573, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December, 1997.

[14] R. Braden. Extending TCP for transactions — concepts. Internet RFC-1379, July 1994.

[15] Edmund Burke. Two letters addressed to a member of the present parliament, on the proposals for peace with the regicide directory of france, letter I. In Edward John Payne and Francis Canavan, editors, *Edmund, Select Works of Edmund Burke, and Miscellaneous Writings*, volume 3. Liberty Fund, Inc., 1999.

[16] Lewis Carroll. *Alice's Adventures In Wonderland*. Project Gutenberg, May 1997.

[17] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Co., Reading, MA, 1988.

[18] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

[19] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8(4):391–401, July 1982.

[20] G. Chapman, J. Cleese, E. Idle, T. Jones, T. Gilliam, and M. Palin. Drinking philosophers. University of Woolloomooloo, November 1970.

[21] S. Chaudhuri and P. Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. In *10th International Workshop on Distributed Algorithms*, October 1996. To appear in *Lecture Notes in Computer Science,* Springer Verlag.

[22] Anna E. Chefter. A simulator for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1998.

[23] Oleg Cheiner. Implementation and evaluation of an eventually-serializable data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1997.

[24] Oleg Cheiner and Alex Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. In M. Mavronicolas, M. Merritt, and N. Shavit, editors, *Networks in Distributed Computing*, volume 45 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–72. American Mathematical Society, 1999.

[25] Oleg Cheiner and Alex A. Shvartsman. Implementation of an eventually serializable data service. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Distributed Computing*, Puerta Vallarta, Mexico, June 1998. (Short paper).

[26] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *Proceedings of the 17th ACM Annual Symposium on Distributed Computing*, June 1998.

[27] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory — practical tools for specification, simulation, verification and implementation of concurrent systems. In *Specification of Parallel Algorithms. DIMACS Workshop*, pages 75–89. American Mathematical Society, 1994.

[28] R. Cleaveland, J. Parrow, and B. U. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1), 1993.

[29] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.

[30] Laura G. Dean. Improved simulation of Input/Output automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2001.

[31] Piotr Dembinski. Semantics of timed concurrent systems. *Fundamenta Informaticae*, 29:27–50, 1997. IOS Press.

[32] Roberto DePrisco, Alan Fekete, Nancy Lynch, and Alex Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the 17th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 227–236, Puerto Vallarta, Mexico, June-July 1998.

[33] M.C.A. Devillers. Translating IOA automata to PVS. Preliminary Research Report CSI-R9903, Computing Science Institute, University of Nijmegen, the Netherlands, feb 1999.

[34] M.C.A. Devillers, W.O.D. Griffioen, J.M.T. Romijn, and F.W. Vaandrager. Verification of a leader election protocol — formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.

[35] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G. V. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55, Berlin, June 1992. Springer-Verlag.

[36] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[37] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 300–309, Philadelphia, PA, May 1996.

[38] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data service. *Theoretical Computer Science*, 220(1):113–156, June 1999. Special Issue on Distributed Algorithms.

[39] Alan Fekete, M. Frans Kaashoek, and Nancy Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1):35–69, January 1998.

[40] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

[41] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[42] Benjamin Franklin. When we launch our little fleet of barques. *The Federal Gazette*, March 1790.

[43] Stephen Garland, Nancy Lynch, Joshua Tauber, and Mandana Vaziri. IOA user guide and reference manual. Technical Report MIT/LCS/TR-961, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 2004. URL `http://theory.lcs.mit.edu/tds/ioa/manual.ps`.

[44] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[45] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998. URL `http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[46] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, USA, 2000.

[47] Kenneth J. Goldman. *Distributed Algorithm Simulation using Input/Output Automata*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1990. Also,[48].

[48] Kenneth J. Goldman. Distributed algorithm simulation using input/output automata. Technical Report MIT/LCS/TR-490, MIT Laboratory for Computer Science, Cambridge, MA, September 1990. Also, PhD Thesis [47].

[49] Kenneth J. Goldman. Highly concurrent logically synchronous multicast. *Distributed Computing*, 6(4):189–207, 1991.

[50] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, September 1995.

[51] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing, editors. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag Texts and Monographs in Computer Science, 1993.

[52] J.V. Guttag, J.J Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, pages 24 –35, September 1985.

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

[54] M. Hayden and R. van Renesse. Optimizing layered communication protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, November 1996.

[55] Mark Hayden. *Ensemble Reference Manual*. Cornell University, 1996.

[56] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings of the International Workshop on TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165, Nijmegen, The Netherlands, 1994. Springer-Verlag. Full version available as Report CS-R9420, CWI, Amsterdam, March 1994.

[57] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[58] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Fifth International Conference, (TACAS'99), Amsterdam, the Netherlands, March 1999*, volume 1579 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 1999.

[59] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, United Kingdom, 1985.

[60] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, New Jersey, 1991.

[61] Carter B. Horsley. "frank gehry, architect" at the solomon r. guggenheim museum. *The City Review*, may 01. `http://www.thecityreview.com/gehgug.html`.

[62] Information Processing Systems Open Systems Interconnection. ESTELLE — A formal description technique based on an extended state transition model, 1989.

[63] Samuel Johnson. Adventurer 138. In *The Works of Samuel Johnson*, volume 11. F. C. and J. Rivington, 1823.

[64] S. Kalvala. A formulation of TLA in Isabelle. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971, pages 214–228, Aspen Grove, Utah, USA, 1995. Springer-Verlag.

[65] Dilsun Kırlı Kaynar, Anna Chefter, Laura Dean, Stephen Garland, Nancy Lynch, Toh Ne Win, and Antonio Ramırez-Robredo. The IOA simulator. Technical Report MIT-LCS-TR-843, MIT Laboratory for Computer Science, Cambridge, MA, July 2002.

[66] Dilsun Kırlı Kaynar, Nancy Lynch, Sayan Mitra, and Christine Robson. Design for TIOA modeling language. Manuscript in progress, 2004.

[67] Dilsun Kırlı Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed I/O automata. Technical Report MIT-LCS-TR-917a, MIT Laboratory for Computer Science, Cambridge, MA, April 2004.

[68] Donald Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion, March 1977. Manuscript.

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

[70] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing Journal*, 6(5):580–584, September/October 1994.

[71] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[72] Leslie Lamport. *Specifying concurrent systems with TLA*, pages 183–247. Number 173 in Computer and Systems Sciences. IOS Press, 1999.

[73] Leslie Lamport, 2000. personal communicaton.

[74] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, December August-September.

[75] Gérard Le Lann. Distributed systems - towards a formal approach. In Bruce Gilchrist, editor, *Information Processing 77* (Toronto, August 1977), volume 7 of *Proceedings of IFIP Congress*, pages 155–160. North-Holland Publishing Co., 1977.

[76] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *In Springer International Journal of Software Tools for Technology Transfer*, pages 1(1+2), 1997.

[77] Aldo Leopold. *Round River: From the Journals of Aldo Leopold.* Oxford University Press, 1993.

[78] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[79] Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1995.

[80] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques* (FORTE'94, Berne, Switzerland, October 1994), pages 259–273. Chapman and Hall, 1995.

[81] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[82] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[83] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[84] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, British Columbia, Canada, August 1987.

[85] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November 1988.

[86] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[87] Panayiotis Mavrommatis, July 2004. personal communication.

[88] Panayiotis Mavrommatis and Chryssis Georgiou. Running distributed algorithms using the IOA toolkit. Manuscript, July 2004.

[89] José Meseguer. Conditional rewriting logic as an unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[90] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, United Kingdom, 1989.

[91] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technishen Universität München, June 1998.

[92] Olaf Müller and Tobias Nipkow. Traces of I/O automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 580–594. Springer, 1997.

[93] E. Kim Nebeuts. *H. Eves Return to Mathematical Circles*. Prindle, Weber and Schmidt, 1988.

[94] Atish Dev Nigam. Enhancing the IOA code generator's abstract data types. Manuscript, 2001.

[95] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 101–119, Båstad, Sweden, 1995. Springer-Verlag LNCS 996.

[96] Jonathan S. Ostroff. A visual toolset for the design of real-time discrete event systems. *IEEE Transactions on Control Systems Technology*, 5(3), May 1997.

[97] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs, I. *Acta Informatica*, 6(4):319–340, 1976.

[98] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[99] Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13, September 1982.

[100] Tsvetomir P. Petrov, Anna Pogosyants, Stephen J. Garland, Victor Luchangco, and Nancy A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* (FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996), pages 29–44. Chapman & Hall, 1996.

[101] J. Postel. Transmission Control Protocol — DARPA Internet Program Specification (Internet Standard STC-007). Internet RFC-793, September 1981.

[102] J. Antonio Ramırez-Robredo. Paired simulation of I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2000.

[103] Henry Reed. Naming of parts. *New Statesman and Nation*, 1942.

[104] Elvinia Riccobene, Myla M. Archer, and Constance L. Heitmeyer. Applying TAME to I/O automata: A user's perspective. Technical Report NRL.MR.5540–00-8848, Naval Research Laboratory, April 2000.

[105] Christine Margaret Robson. TIOA and UPPAAL. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 2004.

[106] A.W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '95), volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer-Verlag, 1995.

[107] Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, January 1983.

[108] N. Shankar, Sam Owre, and John Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab, SRI Intl., Menlo Park, CA, 1993.

[109] Richard M. Sherman and Robert B. Sherman. A spoonful of sugar. Mary Poppins, 1964.

[110] Mark Smith. Formal verification of communication protocols. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996, pages 129–144. Chapman & Hall, 1996.

[111] Mark Smith. *Formal Verification of TCP and T/TCP*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1997.

[112] Mark Smith. Reliable message delivery and conditionally-fast transactions are not possible without accurate clocks. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Distributed Computing*, pages 163–171, June 1998.

[113] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th International Conference, CAV'93, Elounda, Greece, June/July 1993), volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[114] Edward Solovey. Simulation of composite I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2003.

[115] Ekrem Söylemez. Automatic verification of the timing properties of MMT automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1994.

[116] Sun Microsystems, Inc. Java remote method invocation specification, February 1997.

[117] Sun Microsystems, Inc. Java message service specification, November 1999.

[118] Joshua A. Tauber and Stephen J. Garland. Definition and expansion of composite automata in IOA. Technical Report MIT/LCS/TR-959, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 2004. URL `http://theory.lcs.mit.edu/tds/papers/Tauber/MIT-LCS-TR-959.pdf`.

[119] Michael J. Tsai. Code generation for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 2002.

[120] Mandana Vaziri, Joshua A. Tauber, Michael J. Tsai, and Nancy Lynch. Systematic removal of nondeterminism for code generation in I/O automata. Technical Report MIT/LCS/TR-960, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 2004. URL `http://theory.lcs.mit.edu/tds/papers/Tauber/MIT-LCS-TR-960.ps`.

[121] Toh Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 2003.

[122] Toh Ne Win, Michael Ernst, Stephen Garland, Dilsun Kirli, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 4:1–10, 2003.

[123] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In Laurence Pierre and Thomas Kropf, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, 1999.