Compiling IOA without Global Synchronization

Joshua A. Tauber, Nancy A. Lynch, Michael J. Tsai
MIT Computer Science and Artificial Intelligence Laboratory
josh,lynch,mjt@csail.mit.edu

ABSTRACT
This paper 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.

1 INTRODUCTION
Reasoning about and building distributed systems is notoriously difficult. I/O automata provide a simple mathematical basis for formally modeling and understanding distributed systems. Using a rich set of proof techniques, I/O automata have been used to verify a wide variety of distributed systems and algorithms and to express and prove several impossibility results. IOA is a formal language for describing I/O automata that has been introduced to promote I/O automata-based techniques and to support an integrated software development environment for distributed systems. This environment, the IOA toolkit, is intended to support 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.

This paper presents a strategy for compiling distributed systems specified in IOA into Java programs running on a group of networked workstations. The IOA toolkit and compiler will enable programmers to write their specifications as a collection of nodes communicating via reliable, FIFO, one-way channels. The program for each node (or a parameterized set of nodes) is submitted individually to the IOA compiler. The resulting collection of Java programs (one for each node) runs without any global synchronization. Each node program runs in its own Java Virtual Machine (JVM) on the designated host. Compilation adds little communication overhead (e.g., low-level acknowledgments added by MPI). The compilation process imposes no global synchronization overhead.

We claim that under certain conditions the compilation method preserves the safety properties of the IOA program in the generated Java code. This result is based on the assumptions that our model of network behavior is accurate, and that our hand-coded datatype library correctly implements its semantic specification. Moreover, we assume that the system is designed so that its safety properties hold even when inputs to any node in the system are delayed. Our current model of network behavior does not allow for failures. To prove this claim, 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.

Section 2 introduces the input/output automaton model, the IOA language, and the IOA toolkit. Section 3 describes the form an IOA program must have to be admissible for compilation. Section 4 describes how the programmer annotates an IOA program to resolve its inherent nondeterminism. Section 5 describes the translation process. Section 6 presents an argument for the correctness of the compilation method. Section 7 discusses related tools-based approaches to formal methods.

2 IOA LANGUAGE AND TOOLKIT

2.1 Input/Output Automata

I/O automata provide a simple mathematical basis for understanding distributed systems [26, 27]. 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). An action \( \pi \) is enabled in state \( s \) if there is some triple \((s, \pi, s')\) in the transition relation of the automaton. Input actions are required to be enabled in all states.

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 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 simultaneous 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 [30].) 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 defined as a mapping between the states of two automata that is maintained in all reachable states while preserving the external behavior of the automata. 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. To prove a relation is a simulation relation, one demonstrates 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 [23, 25, 10]. A succinct explanation of the model and many of its proof techniques appears in Chapter 8 of [24].

2.2 IOA Language

The IOA language [3] 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. States are represented by the values of variables rather than just by members of an unstructured set. IOA transitions are described in precondition-effect (or guarded-command) style, rather than as state-action-state triples. 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 and programming language, it supports both imperative 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.

---

1 We omit discussion of tasks, which are sets of non-input actions.
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 \((s, \pi)\) may not uniquely determine the following state \(s'\) in a transition relation.

### 2.3 Example: LCR Leader Election

We illustrate IOA by describing the LeLann-Chang-Roberts (LCR) leader election algorithm as a composition of process and channel automata.

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 1 shows a Channel automaton describing communication channels by which processes can send messages. This automaton represents a reliable communication channel, which neither loses nor reorder messages in transit. The automaton is parameterized by the two 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 variables) whose values are fixed by the values of the automaton’s parameters. The state of the automaton Channel consists of a buffer, which is a sequence of messages (i.e., an element of type \(\text{Seq}[\text{Int}]\)) initialized to the empty sequence \(\text{[]}\). The operators on sequences used are: \(\text{[]}\) (the empty sequence), \(\text{+}\) (append), \(\text{head}\) (the first element of the sequence), and \(\text{tail}\) (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: Node automaton Process](image)

Figure 2 describes a participating LCR process, which is parameterized by the name of the process and number of processes participating in the election. The type declaration on the first two lines of Figure 2 declares Status to be an enumeration of the values idle, voting, elected, and announced. The automaton Process 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 to idle. The input action vote sets status to voting to indicate that an election has begun. The input action receive\((m, \text{const mod}(i-1, \text{ringSize}), \text{const } i)\) may result in three different transitions depending on how the message \(m\) received from the Process 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 \(i - 1 \mod \text{ringSize}\) by the action signature. The automaton has two kinds of output actions: send\((m, i, \text{mod}(i+1, \text{ringSize}))\), which sends a message in pending to the Process automaton to the right, and leader\((i)\), which announces successful election.

The full LCR leader election algorithm is described in Figure 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 Process automaton, \(P[i]\), and one Channel 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 $\text{send}(m, i, \text{mod}(i+1, \text{ringSize}))$ of $P[i]$ are identified with the input actions $\text{send}(m, i, \text{mod}(i+1, \text{ringSize}))$ of $C[i]$, and the input actions $\text{receive}(m, i, \text{mod}(i+1, \text{ringSize}))$ of $P[i]$ are identified with the output actions $\text{receive}(m, \text{mod}(i-1, \text{ringSize}), i)$ of $C[\text{mod}(i-1, \text{ringSize})]$, which is $\text{Channel}(\text{mod}(i-1, 10), i)$. 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.

### 2.4 IOA Toolkit

The IOA language was created as the first step in building an integrated software development environment for distributed systems. The IOA language enables designers to precisely specify automata, their properties, and their relations 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. 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.

In addition to verification tools, the IOA toolkit includes a compiler that translates a restricted subset of IOA programs into Java. The resulting code is intended to run on a collection of workstations communicating using standard networking protocols. In systems without a compiler, actually building a distributed system remains outside the model. A human has to 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, there is a disconnect between the properties of the specification and those of the actual running code. One goal in designing the IOA language and toolkit is to eliminate this gap in existing formal methods. This paper describes the design 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.

### 3 STRUCTURING THE DESIGN

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. The goal of the IOA compiler is to generate 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 synchronization among processes running on different machines, thus reducing the possible sources of concurrency. This approach was taken, for example, by Cheiner and Shvartsman [6, 7]. 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) [13, 4]. 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.

#### 3.1 Imperative IOA programs

As mentioned in Section 2.2, 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. 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 extension to the NDR language discussed in Section 4.)

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 like those specified in Figure 1.

However, the compiled code must still interface with MPI. Therefore, we define auxiliary IOA automata to mediate between MPI and the algorithm automaton. The recvMediator 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 \( 2N \) mediator automata (one for each of its channels). Figure 4 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.

3.3 Console Interface

We divide the input actions of an algorithm automaton into two categories. Network inputs are the actions associated with the receive input transition which connects with incoming channels. Console inputs are all the other input actions.

The I/O automaton model requires that input actions are always enabled. However, Java programs are not input enabled. Console inputs might not be handled immediately upon arrival. Our implementation buffers inputs until the program can process them. Therefore, we require that the IOA system submitted for compilation be designed so that

Figure 5: A node automaton composed of a buffer automaton, an algorithm automaton, and mediator automata is the input to the IOA compiler.

its safety properties hold even when console inputs to any node in the system are delayed. Furthermore, if inputs are entered too quickly at the console, the console buffer may overflow and inputs may be lost. For all inputs to be handled, we require the rate of console inputs to be limited.

Specifically, the programmer must write IOA programs so that the algorithm is correct even when each node automaton is composed with a buffer automaton. Each buffer automaton interface mimics the console input actions of its corresponding algorithm automaton. The buffer automaton has an input action, an internal action, and an output action corresponding to each console input action of the algorithm automaton. The effect of the buffer input action is to place a representation of the algorithm input action invocation in a finite console buffer. If inputs occur too quickly and the console buffer fills, further invocations are ignored. The internal action is enabled when such an invocation is in the console buffer and has the effect of moving it to from the console buffer to an unbounded internal delay queue. The corresponding buffer output action is enabled when the delay queue is not empty and has the effect of removing the invocation from the delay queue.

The buffer automaton for the LCR example is quite simple. The console input action of the LCR automaton is the unparameterized action vote. So the buffer automaton LCRBuffer has three actions in its signature and two queues in its state. The input action vote adds an action to the first queue. The internal action moves the invocation from the first queue to the second. The output action removes an action from the second queue. The code for LCRBuffer is not shown.

3.4 Composition

The completed design is called the composite node automaton and is described as the composition of the algorithm automaton with its associated mediator and buffer automata.
A composer tool [34] expands this composition into a new, equivalent IOA program in primitive form (i.e., without any components statements). 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 below) 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 5 depicts the composition that produces a node automaton.

\[ \text{automaton LCRNode}(i \in \mathbb{N}) \]

\[ \text{components} \]

\[ P : \text{Process}(i, 10); \]

\[ \text{RM}[j : \mathbb{N}] : \text{recvMediator}(i, j) \]

\[ \text{where } 0 \leq j \land j < 10; \]

\[ \text{SM}[j : \mathbb{N}] : \text{sendMediator}(i, j) \]

\[ \text{where } 0 \leq j \land j < 10; \]

\[ B : \text{LCRBuffer}; \]

Figure 6: IOA specification for one node of an LCR system

For the LCR example, the composite node automaton is shown in Figure 6. In that figure, the composite node LCRNode is the result of composing one instance of the Process automaton with one instance of the LCRBuffer automaton and ten instances each of the recvMediator and sendMediator interface automata. The resulting composite node program is parameterized by its name \( i \). The primitive form of the LCRNode automaton output by the composer tool (the node automaton for the LCR system) is not shown.

4 RESOLVING NONDETERMINISM

The IOA language is inherently nondeterministic. In any state of the automaton, many actions may be enabled and, for any action, several following states may result. Developing a method to resolve nondeterminism was the largest conceptual hurdle in the 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 this implicit nondeterminism is called scheduling. 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 (with some help).

4.2 Choosing

As mentioned in Section 2.2, 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

\[ \text{num} := \text{choose } n : \mathbb{N} \text{ where } 0 \leq n \land n < 3 \]

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 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.

It is important to note that schedules and determinator blocks do not change the semantics of IOA programs but rather just annotate them with the mechanisms to resolve nondeterminism. Furthermore, a schedule or determinator block can 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. Nothing in the NDR annotations can cause the program to violate
those safety guarantees. NDR annotations are responsible only for making the program live.

5 TRANSLATING IOA INTO JAVA

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, 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. The automaton class is organized around a main loop derived from the NDR schedule annotation to the IOA program. A second thread processes input actions, placing them in a buffer as they arrive.

Below we discuss the elements of the generated automaton subclass. Section 5.1 discusses the state of the generated program. Section 5.2 discusses the translation of individual IOA operators into Java. Section 5.3 discusses the translation of the main loop. Section 5.4 discusses how the compiler translates the parts of a transition into Java constructs.

5.1 Translating State

Each state variable of the IOA program is translated into a member variable of the generated Java automaton class. These state variables are initialized to the initial values of the IOA program. If the state variable is initialized with a choose, a corresponding determinator block is translated. The classes implementing the types of these variables must be included in a datatype library.

5.2 Translating Datatypes

IOA has been designed to work closely with the Larch Shared Language (LSL) [20, 19]. 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 Java.

Following our philosophy of giving the difficult tasks to humans, we require the programmer to write datatype implementation classes by hand. Each datatype (e.g., Boa, Int, or Mset[__]) is implemented as a Java class. Each operator (e.g., __>__, Int, Int → Boa or size: Mset[__] → Int) is implemented as a method by some datatype implementation. Notice that 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 datatypes and operators to map to the associated implementation class and its methods. The mapping between datatypes and operators and implementation classes and methods is maintained in a datatype registry [35, 37].

The IOA toolkit includes a standard library of implementation classes for the standard language datatype. These include simple datatypes like naturals, integers, and booleans, compound datatypes like arrays, maps, sets, and sequences, and shorthand types like 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 [35] 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 [29].

Since the IOA framework focuses on correctness of the concurrent, interactive aspects of programs rather than on 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.

5.3 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. Tsai’s thesis [35] gives the specifics of translating NDR control structures into Java. At run time, the generated program starts from a unique initial state and iterates the loop that selects a method (transition definition) together with a set of parameter values to execute.

5.4 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 where clause evaluates to false, the transition is not executed and the schedule loop iterates. If they both 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. IOA choose statements are compiled by translating their associated NDR determinator blocks.
5.4.1 Translating MPI Transitions

In our design, the IOA interface to MPI is specified as a set of special transition definitions. The definition of these transitions is fixed inside the mediator automata sendMediator and recvMediator. These transitions are designed to mirror the corresponding Java calls used to invoke MPI. We use only four of the (myriad) methods provided by MPI:

- **Isend** sends a message to a specified destination and returns a handle to name the particular send.
- **test** tests a handle to see if particular send has completed (i.e., has freed up memory for another send).
- **Iprobe** polls to see if an incoming message is available.
- **recv** returns a message when available.

The first three of these are non-blocking: **recv** blocks until a message is available. Our implementation calls **recv** only when we know it will not block (i.e., after **Iprobe** has returned true).

In IOA, we define two transitions for each of these Java methods: one transition for the call and one for the return. The headers for these transitions are shown in Figures 7 and 8. In those figures, the **Handle** type is used to name particular send instances. So, **resp_Isend** returns a handle **h** that can be used by subsequent calls to **test**. The boolean **flag** returned by **resp_test** indicates whether the cited send has completed. The boolean **flag** returned by **resp_Iprobe** indicates whether a message is available. The message **m** itself is returned by **resp_recv**.

```
output Isend(m: M, i: Int, j: Int)
output test(h: Handle, i: Int, j: Int)
input resp_Isend(h: Handle, i: Int, j: Int)
input resp_test(flag: Bool, i: Int, j: Int)
```

Figure 7: MPI transition headers derived from sendMediator

The compiler recognizes these four pairs of corresponding transition definitions and treats them as special cases. Rather than generating two methods for the effects of a pair, the compiler generates a single method that places the relevant MPI method invocation between the translations of the effects of the output and input.

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. We use only MPI calls that we know will not block so the effect as a whole will not block. (We call **recv** only after confirming a message is available.) 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.

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

Figure 8: MPI transition headers derived from recvMediator

5.4.2 Translating Buffer Transitions

The buffer input and output actions described in Section 3.3 are another special case in our translation. Since input actions are not locally controlled, the input and internal actions are translated into their own thread. The output actions are composed with the algorithm input actions and hidden (become internal). These actions are translated into the main automaton thread. The methods implementing the internal actions share access to the delay queue across the thread boundary. To prevent corruption of the queue, the compiler uses the Java **synchronize** construct to protect queue accesses. Note this synchronization is local to the single node program.

6 TRANSLATION CORRECTNESS

We now argue 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 implements the IOA system design submitted for compilation. Schematically, the IOA compiler preserves the behavior at the boundary between the System and Environment automata shown in Figure 9. Formally, Theorem 1 in Section 6.4 asserts that the externally visible behaviors of the compiled system are a subset of the externally visible behaviors of the specification automata. For this result to hold, we assume that our model of network behavior is accurate (as discussed below), and that our hand-coded datatype library correctly implements its LSL specification (as mentioned in Section 5.2).

Notice that the correctness condition is global. We require that the system as a whole preserves external behaviors, not individuals nodes. That is, we must show that the multi-threaded Java programs running on multiple concurrently operating nodes (and not using any global synchronization) preserve the appearance of the sequential execution model of the global system I/O automaton. Our approach is to model the compiled Java code as itself being an I/O automaton which takes many small atomic steps that may be interleaved across threads and nodes. We then define a function mapping the states of the I/O automaton denoted by the source program to the states of our model of the Java program. We prove the function is a refinement mapping by showing a step correspondence.
6.1 MPI

While the running system links to actual MPI libraries, we model the behavior of these libraries as an I/O automaton MPI as denoted by an IOA program MPIAut (not shown). MPIAut makes explicit all our assumptions about the behavior of the network. For example, MPI channels deliver messages in order, without loss or duplication.

As described in Section 5.4.1, we model the four MPI methods our design invokes as pairs of input and output actions. MPIAut details the behaviors and interactions of these methods. For example, MPIAut 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_Iprobe actions are guaranteed to become enabled in a finite number of steps even if no other inputs to MPIAut occur in the execution.

6.2 MacroSystemAut

The complete system designed by the IOA programmer consists of MPIAut and the scheduled IOA programs for all the nodes. We refer to the combination of these programs as the MacroSystem program. Each of these IOA programs denotes an I/O automaton, and the combination denotes the composition of these automata. We call this composition MacroSystemAut. We also refer to the individual node program at each host as Ni and the individual automaton it denotes as Ni.

Each transition definition T in Ni defines a set of state-action-state triples in MacroSystemAut. The node programs also yield additional structure for the node automata. The precondition defines a set of states prestatesTi. The effects define a computable function fT from states to states restricted to the domain prestatesTi. Since MacroSystem is scheduled, the state includes a special PC variable as described in Section 4.1 and the function of each transition definition updates the PC. Figure 10 shows the definition of the MacroSystem program for the LCR example.

```plaintext
automaton LCRSystem components
    Ni[i]: LCRNode(i:Int) where 0 ≤ i ∧ i ≤ 10;
    N[i][j]: MPI(i,j:Int) where 0 ≤ i ∧ i ≤ 10
                        ∧ 0 ≤ j ∧ j ≤ 10
```

Figure 10: LCRSystem automaton using MPI channels

6.3 µSystemAut

We model the compiled Java code itself as an I/O automaton µSystemAut which takes many small atomic steps that may be interleaved across threads and nodes. For each step MacroSystemAut takes, µSystemAut takes a sequence of micro-steps. 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. Each node automaton is compiled in such a way that if a sequence of these micro-steps executes without interruption or interleaving, the cumulative effect of the sequence corresponds to the effect of the corresponding action of MacroSystemAut. By stating and proving Theorem 1 below, we are asserting that even in the presence of interleaving, the system behaves correctly.

Like MacroSystemAut, µSystemAut is an I/O automaton denoted by an IOA program µSystem composed of the MPIAut program and IOA programs for each node automaton in the system. However, the IOA node programs in µSystem are not the node programs in MacroSystem. Rather, each node automaton in µSystem is derived from a corresponding component of MacroSystem. This derivation corresponds to the compilation process.

6.3.1 Deriving a micro-node from the macro-node

For each node automaton Ni in MacroSystem, we define a corresponding micro-node automaton µNi. µNi models the Java code that implements Ni. We model the two Java threads in the node implementation class by giving µNi two program counters (µPC). One µPC controls the execution of the actions in µU derived from the input actions of Ni. The other µPC controls the execution of the actions of µNi derived from the locally controlled actions Ni.

Let Ni and µNi be states of Ni and µNi respectively. For each transition definition T in Ni with effect function fT of Ni, µNi has a sequence of transition definitions T1,T2,… with effect functions f1,f2,… respectively. The precondition of T1 requires that the µPC (for its thread) names T1. Let f∗ be the composition of f1,f2,….

Let s and s’ be states of Ni and u and u’ be states of µNi. Let ū be the projection of u onto the state space of Ni. If s = û, s’ = f(s) and u’ = f∗(ū) then we require that s’ = ū’. Each µPC is a pair. The first element of µPC denotes the macro-action being executed (micro-action sequence) and the second element denotes the micro-action. Each function in the sequence increments the micro-action element of its µPC (in addition to whatever other work it performs). The last function in each sequence schedules the next macro-action by updating the macro-element action and setting the micro-action element to the first micro-action of the corresponding sequence.

Note, while each thread steps sequentially through a sequence of micro-actions, the micro-actions of different threads in µSystemAut can be interleaved with those of different threads (either at the same node or at others) or with steps of the MPIAut.

Note, IOA programs µSystem and Ni are only conceptual. No such IOA programs are ever produced. The method of deriving µSystem from MacroSystem gives a correctness condition for the relevant characteristics of the IOA compiler. That is, if µSystem is an accurate model of the code generated by the compiler, the compilation process preserves safety properties of the submitted automaton.

6.3.2 Locking the delay queue

We model the synchronized methods described in Section 5.4.2 by saying that the first micro-step in the sequence for the corresponding action grabs a lock on the delay queue. The last micro-step releases it. In general we do not specify
the granularity of the micro-steps or the micro-effect functions \( f_i \). However, we do require that there are special micro-steps to grab and release a lock. That is, locking must be atomic with respect to thread interleaving. If another action already has the lock, the function resets the \( \mu PC \) for its thread to itself, in effect spinning on the lock. Note, this spinning only blocks the thread attempting to grab the lock. The lock itself is represented as an additional state variable of the \( \mu N_i \). So locks are local, not global.

### 6.4 Correctness Theorem

The theorem for correctness of the code generator says that the externally visible behaviors of \( \mu SystemAut \) are a subset of those of \( MacroSystemAut \). Thus, if \( \mu SystemAut \) correctly models the generated Java code, the compiled system will exhibit only behaviors specified by the system designer.

**Theorem 1** The traces of \( \mu SystemAut \) are a subset of the traces of \( MacroSystemAut \).

We prove Theorem 1 by demonstrating a refinement mapping from \( \mu SystemAut \) to \( MacroSystemAut \).

#### 6.4.1 History variables

To establish the refinement mapping, we augment the state of each node program \( \pi \), with history variables. In particular, every variable \( v \) in the state of \( \mu System \) — other than the \( \mu PC \) and any locks — is mirrored by a history variable \( u \). The value of \( u \) is initialized to \( v \).

The history variables are updated only by the first or last step of a sequence of micro-actions. In particular, in the last step of locally controlled actions all history variables are assigned the new values of regular variables so that they again mirror the state of \( \pi \). In the first step of input actions, all history variables are assigned the values the regular variables will have at the end of the input action sequence. This is possible because all input actions (both the MPI resp.* actions and the buffer input actions) are deterministic.

#### 6.4.2 Refinement Mapping

We prove Theorem 1 by showing a refinement mapping \( M \) from \( \mu SystemAut \) to \( MacroSystemAut \). We define \( M \) as follows: if \( s \) is a state of \( \mu System \) then \( S = M(s) \) is a state of \( MacroSystem \) such that \( S.v = s.u \) and \( S.PC \) is the macro entry of \( s.\mu PC \).

For every action \( \pi \) enabled in a reachable state \( s \) of \( \mu SystemAut \) we must specify a (possibly empty) sequence of actions \( \alpha \) of \( MacroSystemAut \) such that \( \alpha \) is enabled in state \( S = M(s) \) of \( MacroSystemAut \) and such that if \( s' \) is the state that results from executing \( \pi \) in state \( s \) and \( S' = M(s') \) is the state that results from executing \( \alpha \) in state \( S \), then \( S' = M(s') \). Furthermore, \( \text{trace}(\pi) \) must equal \( \text{trace}(\alpha) \). We define the step correspondence by cases depending on \( \pi \).

If \( \pi \) is neither the first nor last step in a micro-action sequence, \( \alpha \) is empty. Neither the history variables, the macroentry of \( \mu PC \), nor the state of \( MacroSystemAut \) changes. Similarly, if \( \pi \) the last step of an input sequence or the first step on an output sequence, \( \alpha \) is empty.

The interesting cases are when \( \pi \) begins an input micro-action sequence or ends a locally-controlled-action sequence. In those cases, \( \alpha \) is the action of \( MacroSystemAut \) from which \( \pi \) is derived. That is, internal and output actions act “as if” they happen all in the last micro-step and input actions happen “as if” they all happen in the first micro-step.

### 7 RELATED WORK

Goldman’s Spectrum System introduced a formally-defined, purely operational programming language for describing I/O automata [15, 17]. 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 uses a language with formal semantics expressed in terms of I/O automata [18].

Cheiner and Shvartsman experimented with methods for generating code from I/O automaton descriptions [6, 7]. They selected a particular distributed algorithm from the literature (the Eventually Serializable Data Service of Luchango et al. [12]) and generated by hand an executable, distributed implementation in C++ communicating via the Message Passing Interface (MPI [13]). 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.

To our knowledge, no system has yet combined a language with formally specified semantics, verification tools such as automated proof assistants, simulators, and code generators. A number of tools have been based on the CSP model [21]. 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 [32]. The ProBE tool enables the user to “browse” a CSP process by following events that lead from one state of the process to another while resolving nondeterminism.

Cleaveland et al. have developed a series of tools based on the CCS process algebra [28]. The Concurrency Workbench [9] and its successor the Concurrency Factory [8] 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.
A IMPLEMENTATION STATUS

The IOA compiler currently translates unparameterized node automata that run in isolation. The connection between node automata and MPI is currently being implemented. Translation of the states of the node automata that result from composition requires certain extensions to the datatype translation machinery. Currently, the support for compound datatypes of variable dimensions is limited. For example, maps currently can only have a single domain. However, the state of a parameterized automaton is represented in IOA as map from the values of its parameters to the tuple of its state variables. So, for example, the state of a channel that is parameterized by the names of its two end points is a map with two domains (each a node names).

B ABSTRACT CHANNELS

In Section 3.2.1, we stated that IOA programmers can assume the existence of abstract channels like the example Channel automaton. We justify that assumption by showing that a composition of appropriately parameterized MPI and mediator automata implements the abstract channel automaton. Figure 11 shows the IOA specification MPIChannel for that composition. Theorem 2 asserts that MPIChannel implements Channel. The theorem is proved by demonstrating a refinement mapping from MPIChannel to Channel.

automaton MPIChannel(i,j: Int)
components
RM: recvMediator(i,j);
SM: sendMediator(i,j);
M[i,j]: MPI(i,j)
Figure 11: IOA specification for the MPIChannel automaton

Theorem 2 The traces of the MPIChannel are a subset of the traces of Channel.

REFERENCES


