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, a formal language for describing such systems as I/O automata, into Java programs running on a group of networked workstations. The translation works node-by-node, translating IOA programs into Java classes that communicate using the Message Passing Interface. 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 is part of the IOA toolkit which 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 [23, 24]. Complex systems are decomposed into simpler interacting components whose structure can be understood using levels of abstraction and, orthogonally, parallel composition. 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. The IOA toolkit is an integrated software development environment for distributed systems that supports algorithm design, development, testing, and formal verification using automated tools. The toolkit connects I/O automata with both lightweight (syntax checkers, simulators, model checkers [9, 4, 27, 6, 32, 31, 28]) and heavyweight (theorem provers) tools [18, 2].

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 enables programmers to write their specifications at a high level of abstraction, validate the specification using other tools in the IOA toolkit, 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 the actual system implementation.

The translation works node-by-node, translating IOA programs into Java classes that communicate using the Message Passing Interface (MPI) [17, 12]. In order to use the compiler, IOA programmers structure the specification to be compiled 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 require 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.

2. IOA Language and Toolkit

2.1. Input/Output Automata

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.

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 a 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. A succinct explanation of the model appears in Chapter 8 of [22].

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

2.2. IOA Language

The *IOA language* [16] 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. 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 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 a source of *implicit nondeterminism*. 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.

2.3. 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 LCR, a finite set of processes arranged in a ring elect a leader by communicating asynchronously. Each process sends its name to its right neighbor. When a process receives a name greater than its own, the process transmits the received name to the right; other names are discarded. If a process receives its own name, it declares itself the leader [21, 3].

Figure 1 shows a `Channel` automaton describing communication channels through which processes can send messages. This automaton represents a reliable communication channel, which neither loses nor reorders messages in transit. The automaton is parameterized by two indices,

```
automaton Channel(i, j: Int)
  signature
    input  send(m: Int, const i, const j)
    output receive(m: Int, const i, const j)
  states
    buffer: Seq[Int] := {}
  transitions
    input send(m, i, j)
      eff buffer := buffer ⊢ m
    output receive(m, i, j)
      pre buffer ≠ {} ∧ m = head(buffer)
      eff buffer := tail(buffer)
```

Figure 1: Reliable FIFO `Channel` automaton

`i` and `j`, for the processes that communicate by the channel. Its signature consists of input actions, `send(m, i, j)`, and output actions, `receive(m, i, j)`, one for each message `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 initialized to the empty sequence. The operators on sequences used are: `{}` (the empty sequence), ⊢ (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 describes an LCR process, which is parameterized by an index `i`, the number of participating processes, and the `name` of the process. The automaton `Process` has two state variables: `pending` is a multiset of integers and `status` has the enumeration type `Status`. Initially, `pending` contains the name of the process, and `status` is `idle`. The input action `vote` sets `status` to indicate that an election has begun. The input action `receive` 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 its own `name`. These transitions are described in three separate transition definitions. 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 `ringSize` by the action signature. The automaton has two kinds of output actions: `send`, which sends a message in `pending` to the `Process` automaton to the right, and `leader(i)`, which announces successful election.

A 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

```
type Status = enumeration of idle, voting,
                                elected, announced
automaton Process(i, ringSize, name: Int)
  signature
    input  vote
    input  receive(m: Int,
                   const mod(i-1, ringSize),
                   const i)
    output send(m: Int, const i,
                const mod(i+1, ringSize))
                where m ≥ i,
           leader(const i)
  states
    pending: MSet[Int] := {name},
    status: Status := idle
  transitions
    input vote
      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(i, j, name)
      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: Node automaton `Process`

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

Figure 3: `LCR` system automaton using FIFO channels

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 `send(m, i, mod(i+1,ringSize))` of `P[i]` are identified with the input actions `send(m, i, mod(i+1, ringSize))` of `C[i]`, and the input actions `receive(m, i, mod(i+1, ringSize))` of `P[i]` are identified with the output actions `receive(m, mod(i-1,ringSize),i)` of `C[mod(i-1,ringSize)]`.

2.4. IOA Toolkit

In addition to verification tools, the IOA toolkit includes a compiler that translates a restricted subset of IOA programs into Java. In systems without such 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 rest of 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

According to the semantics of IOA, the individual actions of the algorithm, 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 was taken, for example, by Cheiner and Shvartsman [5]. 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, requiring 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*. The target environment for the IOA compiler are networked workstations. Each host runs a Java interpreter connected to a console and communicates with other hosts via MPI. 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.

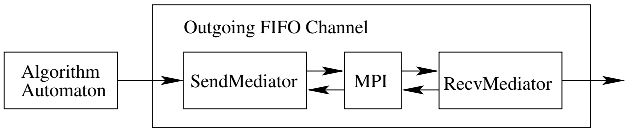

Figure 4: Auxiliary automata mediate between MPI and algorithm automata to yield a reliable, FIFO channel.

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, transition effects cannot include **ensuring** clauses which relate pre-states to post-states declaratively. Throughout the program, predicates must be quantifier free. Loops must have the restrict form that explicitly specifies 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

Systems submitted to the IOA compiler must be described in *node-channel* form. Specifically, a compilable IOA program 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. As in the LCR example, distinct instances of the same algorithm automaton may run at different nodes.

3.2.1. Abstract Channels While code generated by the IOA compiler must interface with MPI, the intricacies of using MPI 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 connects to up to $2N$ mediator automata (one for each of its channels). Figure 4 depicts how a mediator automaton is composed with MPI to create an abstract channel.

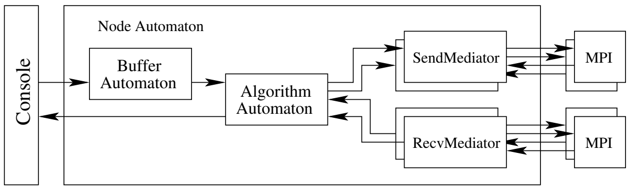

Figure 5: Node automata are submitted to the IOA compiler.

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.

While, the I/O automaton model requires that input actions are always enabled, we require that the safety properties of an IOA system submitted for compilation hold even when console inputs to any node in the system are delayed because console inputs to Java programs might not be handled immediately. Furthermore, if inputs arrive too quickly at the console, the console buffer may overflow and inputs may be lost.

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. The internal action is enabled when such an invocation is in the console buffer and has the effect of moving it 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 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 (see Figure 5). A *composer* tool [30] 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 (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.

```
automaton LCRNode(i, name:Int)
  components
    P: Process(i, 10, name);
    RM[j:Int]: recvMediator(i,j)
                  where 0 ≤ j ∧ j < 10;
    SM[j:Int]: sendMediator(i,j)
                  where 0 ≤ j ∧ j < 10;
    B: LCRBuffer;
```

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

The automaton `LCRNode` (Figure 6) is the composition of 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 is parameterized by its name and index. 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

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 resolving both kinds of nondeterminism is the same: we let the programmer do it.

4.1. Scheduling

Picking a transition to execute includes picking 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. Furthermore, transition preconditions may be arbitrary predicates in first-order logic. Thus, determining, for a given state of an automaton, whether there exists a set of parameters for some transition definition that will enable it in that state is undecidable.

One might imagine that restricting the class of automata accepted for compilation would admit a brute force search-based solution to the scheduling problem. However, it is not obvious how to formulate such restrictions without radically restricting the expressive power of the language. Also, 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, we require the programmer to 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 a *nondeterminism resolution language* (NDR) consisting of imperative programming constructs like those used in IOA effects clauses [27, 31]. The `fire` statement schedules a transition to run and selects the values of its parameters. Schedules may reference, but not modify, automaton state variables.

Conceptually, adding an NDR schedule to an IOA program changes it in three ways. The NDR schedule adds new variables, modifies each transition to use the new variables, and provides a computable *next-action* function of the augmented state. The new state variables consist of a program counter (PC) 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 only if the PC names the action. Second, at the end of the effects the PC is assigned the next action as computed by applying the next-action function to the automaton state [13].

4.2. Choosing

The **choose** statement introduces explicit nondeterminism in IOA. When a **choose** statement is executed, an IOA program selects an arbitrary value from a specified et. For example, the statement

`num := choose n:Int where` $0 \le n \wedge n < 3$

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. In this case, the programmer annotates the **choose** statement with an NDR *determinator block*. The `yield` statement specifies the value to resolve a nondeterministic choice. Determinator blocks may reference, but not modify, automaton state variables.

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. Standard libraries include this generic class, support for console interactions, MPI initialization, and 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.

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) [10]. 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.

Therefore, the IOA toolkit includes a standard library of hand-coded implementation classes for the standard language datatypes. 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 (see [31]). Each new datatype (*e.g.*, `Tree[__]`) must be implemented by hand as a Java class. Each operator (*e.g.*, `Height: Tree[__]` $\rightarrow$ `Nat]`) is implemented as a method by some datatype implementation. Notice that since operators signatures may reference more than one type, 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 [31, 32]. The programmer specifies at compile time which datatypes to load, and the datatype registry is initialized appropriately [26].

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 for such 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 [31] 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`. The effects clause is executed only if the precondition and **where** clause evaluate to true.

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 the 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 it will not block (*i.e.*, after **Iprobe** has returned true).

We define two IOA transitions for each of these Java methods: one for the call and one for the return (see Figures 7 and 8). 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: `sendMediator` MPI transitions

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. As a result, the input half of the pair (the `resp_*` transition) does not need to be scheduled. The input is executed without returning to the schedule loop when the MPI call returns.

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 run in their own thread. The output actions are composed with the algorithm input actions and hidden (become internal). These actions run in 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 a single node program.

```
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: `RecvMediator` MPI transitions

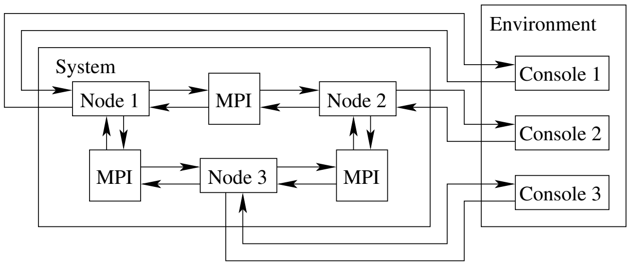

Figure 9: A compiled IOA system consists of a composition of node and MPI automata interacting with the environment.

6. Translation Correctness

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 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.

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 $\mathtt{N}_i$ and the individual automaton it denotes as N_i.

Each transition definition `T` in $\mathtt{N}_i$ 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 $prestates_{\mathrm{T}}$. The effects define a computable function f_{T} from states to states restricted to the domain $prestates_{\mathrm{T}}$. 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. (`Name` is an unspecified integer function.)

```
automaton LCRSystem
  components
    N[i,name]: LCRNode(i,name:Int)
      where 0 ≤ i ∧ i ≤ 10 ∧ name = Name(i);
    M[i,j]: MPI(i,j:Int)
      where 0 ≤ i ∧ i ≤ 10 ∧ 0 ≤ j ∧ j ≤ 10
```

Figure 10: `LCRSystem` automaton using MPI channels

6.3. $\mu SystemAut$

We model the compiled Java code itself as an I/O automaton $\mu SystemAut$ which takes many small atomic steps that may be interleaved across threads and nodes. For each step $MacroSystemAut$ takes, $\mu SystemAut$ 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 $MacroSystemAut$, $\mu SystemAut$ 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 $\mathtt{N}_i$ in MacroSystem, we define a corresponding micro-node automaton $\mu\mathtt{N}_i$. $\mu\mathtt{N}_i$ models the Java code that implements $\mathtt{N}_i$. We model the two Java threads in the node implementation class by giving $\mu\mathtt{N}_i$ two program counters (μPC). One μPC controls the execution of the actions in $\mu\mathtt{N}$ derived from the input actions of $\mathtt{N}_i$. The other μPC controls the execution of the actions of $\mu\mathtt{N}_i$ derived from the locally controlled actions $\mathtt{N}_i$.

Let N_i and μN_i be the I/O automata denoted by $\mathtt{N}_i$ and $\mu\mathtt{N}_i$, respectively. For each transition definition of $\mathtt{T}$ with effect function f_{T} of $\mathtt{N}_i$, $\mu\mathtt{N}_i$ has a sequence of transition definitions $\mathtt{T}_1, \mathtt{T}_2, \ldots$ with effect functions $f_1, f_2, \ldots$, respectively. The precondition of $\mathtt{T}_i$ requires that the μPC (for its thread) names $\mathtt{T}_i$. Let f^* be the composition of $f_1, f_2, \ldots$.

Let s and s' be states of N_i and u and u' be states of μN_i. Let $\dot{u}$ be the projection of u onto the state space of N_i. If $s = \dot{u}$, $s' = f(s)$ and $u' = f^*(u)$ then we require that $s' = \dot{u}'$.

Each μPC is a pair. The first element of μPC denotes the micro-action sequence being executed and the second element denotes the index in that sequence. Each function in the sequence increments the index of its μPC (in addition to whatever other work it performs). The last function in each sequence schedules the next macro-action by pointing its μPCto the first element of the corresponding sequence.

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

Note, IOA programs μSystem and $\mathtt{N}_i$ 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 locks. That is, locking is atomic with respect to thread interleaving. If another action already has the lock, the function resets the μPC 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 is represented as a state variable of the μN_i. So locks are local, not global.

6.4. Correctness Theorem

The compiler correctness theorem asserts that, if $\mu SystemAut$ correctly models the generated Java code, the compiled system will exhibit only behaviors specified by the system designer in $MacroSystemAut$.

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

Theorem 1 is by proved demonstrating a refinement mapping from $\mu SystemAut$ to $MacroSystemAut$ in [29]. The proof is omitted for lack of space.

7. Related Work

Goldman's Spectrum System introduced a formally-defined, purely operational programming language for describing I/O automata [19]. 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 [11].

Cheiner and Shvartsman experimented with methods for generating code from I/O automaton descriptions [5]. They selected a particular distributed algorithm from the literature (the Eventually Serializable Data Service of Luchangco *et al.* [7]) and generated by hand an executable, distributed implementation in C++ communicating via MPI. 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 or directly connect to any verification tools.

To our knowledge, no system has yet combined a language with formally specified semantics, automated proof assistants, simulators, and compilers. Several tools have been based on the CSP model [20]. The semantics of the Occam parallel computation language is defined in CSP [1]. While there are Occam compilers, we have found no evidence of verification tools for Occam programs. Formal Systems, Ltd., developed a machine-readable language for CSP. The FDR model checker allows the checking of a wide range of general safety and liveness properties of CSP models [8]. The ProBE tool enables the user to "browse" a CSP process by following events from one state to another while resolving nondeterminism.

Cleaveland *et al.* have developed a series of tools based on the CCS process algebra [25]. The Concurrency Workbench [14] and its successor the Concurrency Factory [15] 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.

References

[1] INMOS Ltd: occam Programming Manual, 1984.

[2] A. Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, EECS, MIT, Cambridge, MA, Sep 2001.

[3] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *CACM*, 22(5):281–283, May 1979.

[4] A. Chefter. A simulator for the IOA language. Master's thesis, EECS, MIT, Cambridge, MA 02139, May 1998.

[5] O. Cheiner and A. Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. In M. Mavronicolas *et al.*, editor, *Networks in Distributed Computing*, volume 45 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–72. AMS, 1999.

[6] L. Dean. Improved simulation of Input/Output automata. Master's thesis, EECS, MIT, Cambridge, MA, Sep 2001.

[7] A. Fekete *et al.*. Eventually-serializable data services. In *PODC*, pages 300–309, Philadelphia, PA, May 1996.

[8] A. Roscoe *et al.*. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In Ed Brinksma *et al.*, editor, *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer-Verlag, 1995.

[9] D. Kaynar *et al.*. The IOA simulator. Technical Report MIT-LCS-TR-843, MIT LCS, Cambridge, MA 02139, Jul 2002.

[10] J. Guttag *et al.*, editor. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag Texts and Monographs in Computer Science, 1993.

[11] K. Goldman *et al.*. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, Sep 1995.

[12] M. Baker *et al.*. mpiJava: A Java interface to MPI.

[13] M. Vaziri *et al.*. Systematic removal of nondeterminism for code generation in i/o automata. Technical Report MIT/LCS/TR-960, MIT LCS, Cambridge, MA, Jul 2004.

[14] R. Cleaveland *et al.*. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1), 1993.

[15] R. Cleaveland *et al.*. 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.

[16] S. Garland *et al.*. IOA user guide and reference manual. Technical Report MIT/LCS/TR-961, MIT LCS, Cambridge, MA, Jul 2004.

[17] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[18] S. Garland and N. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, MIT LCS, Cambridge, MA, Aug 1998.

[19] K. Goldman. Highly concurrent logically synchronous multicast. *Distributed Computing*, 6(4):189–207, 1991.

[20] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, United Kingdom, 1985.

[21] G. Le Lann. Distributed systems - towards a formal approach. In Bruce Gilchrist, editor, *Information Processing 77* (Toronto, Aug 1977), volume 7 of *IFIP Congress*, pages 155–160. North-Holland Publishing Co., 1977.

[22] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, Mar 1996.

[23] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, Vancouver, British Columbia, Canada, Aug 1987.

[24] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, Sep 1989.

[25] R. Milner. *Communication and Concurrency*. Prentice-Hall International, United Kingdom, 1989.

[26] A. Dev Nigam. Enhancing the IOA code generator's abstract data types. Manuscript, 2001.

[27] J. Ramırez-Robredo. Paired simulation of I/O automata. Master's thesis, EECS, MIT, Cambridge, MA, Sep 2000.

[28] E. Solovey. Simulation of composite I/O automata. Master's thesis, EECS, MIT, Cambridge, MA, Sep 2003.

[29] J. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, EECS, MIT, Cambridge, MA, Sep 2004.

[30] J. Tauber and S. Garland. Definition and expansion of composite automata in IOA. Technical Report MIT/LCS/TR-959, MIT LCS, Cambridge, MA, Jul 2004.

[31] M. Tsai. Code generation for the IOA language. Master's thesis, EECS, MIT, Cambridge, MA, Jun 2002.

[32] T. Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, EECS, MIT, Cambridge, MA, May 2003.