13

Using I/O Automata for Developing Distributed Systems

Stephen J. Garland
MIT Laboratory for Computer Science,
545 Technology Square, Cambridge, MA 02139
garland@lcs.mit.edu

Nancy Lynch
MIT Laboratory for Computer Science,
545 Technology Square, Cambridge, MA 02139
lynch@theory.lcs.mit.edu

Abstract

This paper describes a new experimental programming language, IOA, for modeling and implementing distributed systems, plus designs for a set of tools to support IOA programming. The language and tools are based on the *I/O automaton model* for reactive systems, which has been used extensively for research on distributed algorithms. The language supports structured modeling of distributed systems using shared-action composition and levels of abstraction. The tools are intended to support system design, several kinds of analysis, and generation of efficient runnable code.

13.1 Introduction

Distributed systems are required to provide increasingly powerful services, with increasingly strong guarantees of performance, fault-tolerance, and security. At the same time, the networks in which these systems run are growing larger and becoming less predictable. It is no wonder that distributed systems have become very complex.

The best approach to managing the increased complexity of systems involves organizing them in structured ways, viewing them at different levels of abstraction, and designing them as parallel compositions of interacting components. Such structure makes systems easier to understand, build, maintain, and extend, and can serve as the basis for documentation and analysis. However, in order to be most useful, this structure must rest on a solid mathematical foundation. This is obviously necessary if the structure is to support formal methods of constructing or analyzing systems; however, even without formal methods, a mathematical basis is essential for precise understanding.

One reasonable mathematical basis is the *I/O automaton model* [LT87], which has

been used to describe and verify distributed algorithms and to express impossibility results (see, for example, [Lyn96]). Several aspects of this model make it good for such tasks. It is based on set-theoretic mathematics rather than on a particular logic or programming language. I/O automata are nondeterministic, which allows systems to be described in their most general forms. I/O automata have a simple notion of external behavior based on sets of traces of external actions. Moreover, I/O automata can be composed by identifying external actions, in a way that respects external behavior, and pairs of automata can be related using various forms of implementation relations that preserve external behavior. The model supports a rich set of proof methods, including invariant assertion techniques for proving that a property is true in all reachable states, forward and backward simulation methods for proving that one automaton implements another, and compositional methods for reasoning about collections of interacting components.

Also, the model has been extended to a *timed I/O automaton model* [LV96], which allows modeling of timing aspects of distributed systems, including timing assumptions and performance guarantees. Both I/O automata and timed I/O automata can be described using simple guarded-command-style pseudocode (see, for example, [LMWF94, Lyn96]).

Although I/O automata were originally developed for modeling theoretical distributed algorithms, in the past few years they have been used to model practical system components such as distributed shared memory services (for example, [FKL98, FGL$^+$99]), group communication services [FLS97, DFLS98, HLvR99], and standard communication protocols like TCP [Smi97]. This work has resolved ambiguities and contributed proofs that systems meet their specifications. It has led to the discovery of problems, including logical errors in key algorithms in the Orca [BKT92], Horus [vRBM96], and Ensemble [HvR96] systems. Moreover, it has produced I/O automaton pseudocode that is close to actual system code: for example, some I/O automaton pseudocode for the Ensemble system [HLvR99] is similar to the actual ML code that appears in the system implementation.

Because the model and pseudocode have worked well in these case studies, we believe they can be made to play a significant role in developing real distributed systems. In this paper, we describe one way this might work.

Most of the work done so far using I/O automata has been carried out by hand. However, for these methods to play a serious role in system development, they will require computer tool support. So far, tool-based work with I/O automata has consisted mainly of using interactive theorem provers to verify invariant assertions and simulation relations (for example, [Nip89, SAGG$^+$93, PPG$^+$96, Arc97]) for I/O-automaton-based designs. The TAME system [AHS98] provides a high-level interface to the PVS theorem prover [ORR$^+$96] for specifying and proving properties of a timed version of I/O automata. Other tool support for I/O automata includes the SPECTRUM programming language and simulator [Gol90].

13.2 General Design Guidelines

The tool support we are constructing begins with a simple formal language for modeling distributed systems using I/O automata, based on the guarded-command-style pseudocode already in use. Such a language should support the system designer in expressing his/her design at different levels of abstraction, starting with a high-level specification of the required global behavior and ending with a low-level version that can be translated easily into real code. The language should also allow the designer to decompose designs into separable components with clearly defined external behavior.

This language should be supported by tools providing access to a full range of validation methods, including proof using an interactive theorem prover, simulation, and model-checking. These tools should allow designers to reason about properties of their designs at all levels of abstraction, and about relationships between different levels.

However, we would like more than just validation tools: we would also like tools for connecting verified designs to runnable distributed code. (Our experience with systems like Ensemble suggests that such connections are feasible.) Such tools would allow claims and proofs about designs to be carried over automatically to real distributed programs.

In particular, we believe that, with some well-chosen programmer input, real distributed code in a standard programming language like C++, Java, or ML, can be generated automatically from low-level I/O-automaton-based designs. The validation tools should be able to ensure that the final programs are correct, subject to assumptions about externally provided system components (for example, communication services). Runnable distributed code has already been generated by hand translation of some specific I/O-automaton-based distributed algorithm descriptions [Che97, Tau].

A programming environment based on such a language and tools could help mathematicians write distributed programs, and help programmers who are not mathematicians use mathematical methods in their work.

In this paper, we outline our design for such a programming environment and describe our progress on building a research prototype.

As a starting point, we have developed a candidate programming language, the IOA language, designed specifically to describe I/O automata and their relationships. IOA has evolved from the various forms of pseudocode used in previous work; it also uses ideas from SPECTRUM [Gol90]. It allows automata to be described using *transition definitions* (guarded commands) consisting of *preconditions* and *effects*. It allows explicit description of nondeterministic choice, composition, and levels of abstraction. It permits both declarative and imperative system descriptions. Although the IOA language may need to be enhanced later to increase its expressive

power, we think it is a good starting point for developing a good programming environment.

We have also developed designs for a set of tools for validating and transforming IOA descriptions and for generating code from IOA descriptions. We are currently refining the designs and constructing prototypes. Key ideas of the high-level designs involve mechanisms for resolving nondeterminism, support for programming using levels of abstraction, and integration of externally provided system components, by modeling them as automata.

The rest of the paper is organized as follows. Section 13.3 contains a description of the IOA language. Section 13.4 contains an extended example—IOA programs for a toy distributed banking system. Section 13.5 contains a discussion of the language design. Section 13.6 contains an overview of our work on tools to manipulate IOA programs. Section 13.7 contains some conclusions. An earlier version of this paper (with more details) appeared as a technical report [GL98].

13.3 The IOA Language

13.3.1 The I/O Automaton Model

An *I/O automaton* is a labeled state transition system used to model a reactive system. It consists of a set of *actions* π (classified as *input*, *output*, or *internal*), a set of *states* s (including a nonempty subset of *start states*), a set of *transitions* of the form (s, π, s') that specify the effects of the automaton's actions, and a set of *tasks*, which are sets of locally controlled (that is, non-input) actions.† Input actions are *enabled* in all states. The operation of an I/O automaton is described by its *executions* $s_0, \pi_1, s_1, \ldots$, which are alternating sequences of states and actions, and its *traces*, which are the externally visible behavior (sequences of input and output actions) occurring in executions. One automaton is said to *implement* another if all its traces are also traces of the other. I/O automata admit a *parallel composition* operator, which allows an output action of one automaton to be identified with input actions in other automata; this operator respects the trace semantics.

Proof methods supported by the model include invariant assertion techniques for proving that a particular property is true in all reachable states, forward and backward simulation methods for proving that one automaton implements another (see, for example, [LV95]), and compositional methods for reasoning about collections of interacting components. For example, a *forward simulation* from automaton A to automaton B is a relation R between states of A and states of B that satisfies two conditions: (i) each start state of A is R-related to some start state of B, and (ii) for each step (s_A, π, s'_A) of A and each state s_B of B such that $(s_A, s_B) \in R$, there exists an execution fragment (that is, a sequence of steps) of B that "corresponds" to the step in a particular way. Namely, it has the same trace and leads to a state

† Tasks are used primarily to describe liveness; we will mostly ignore them here.

s'_B with $(s'_A, s'_B) \in R$. A summary of the model, its features for expressing system structure, and its proof methods, appears in Chapter 8 of [Lyn96].

The I/O automaton model is similar to the labeled transition system models used to define semantics for process algebraic languages like CSP [Hoa85] and CCS [Mil89]. In particular, those models also define parallel composition in terms of identifying external actions, and have trace-like notions of external behavior. Other languages for describing concurrent systems are based on different types of automata, with different notions of composition and external behavior; for instance, TLA [Lam94] and Unity [CM88] are based on automata that combine via shared variables.

13.3.2 Language Design

The IOA language is designed to allow precise and direct description of I/O automata. Since the I/O automaton model is a reactive system model rather than a sequential program model, the language reflects this fundamental distinction. That is, it is not a standard sequential programming language with some constructs for concurrency and interaction added on; rather, concurrency and interaction are at its core.

The IOA language is designed to support both proving correctness and generating code. This leads to a tension in the design, because the features that make languages suitable for proofs (for example, declarative style, simplicity, and support for nondeterminism) differ from those that make them suitable for code generation (for example, imperative style, expressive power, determinism). Nondeterminism helps verification by allowing designers to validate designs in a general form. A simple language with a declarative style is easiest to translate into the input languages of standard theorem provers and easiest to manipulate in interactive proofs. On the other hand, programmers generally prefer a language with high expressive power. Moreover, a deterministic language with an imperative style is easiest to translate into runnable code.

The starting point for IOA was the pseudocode used in earlier work on I/O automata. This pseudocode contains explicit representations of the parts of an automaton definition (actions, states, transitions, and so on). Transitions are described using *transition definitions* (*TDs*) containing *preconditions* and *effects*. This pseudocode has evolved in two different forms: a *declarative style* (see, for example, [LMWF94]), in which effects are described by predicates relating pre- and post-states, and an *imperative style* (for example, [Lyn96]), in which effects are described by simple imperative programs.

In moving from pseudocode to a formally defined programming language, we made the following design decisions:

- Data types are defined axiomatically, in the style used by Isabelle [Pau93], the

Larch Prover (LP) [GG91, Gar94], PVS [ORR+96] and other theorem provers. This facilitates translation into theorem prover input languages. We provide definitions for built-in data types and allow the programmer to define new types, using the Larch Shared Language (LSL) [GH93].

- TDs can be parameterized and the values of the parameters constrained by predicates that we call "`where` predicates." A TD can have additional `choose` parameters, which are not formally part of the action name, but which allow values to be chosen to satisfy the precondition and then used in describing the effect.
- Since neither the declarative style nor the imperative style for describing TD effects is adequate for all purposes, we allow both, either separately or in combination. Thus, a TD effect may be described entirely by a program, entirely by a predicate, or by a combination: a program that includes explicit nondeterministic choices, followed by a predicate that constrains these choices.
- Imperative descriptions of effects are kept simple, consisting of (possibly nondeterministic) assignments, conditionals, and simple bounded loops. This simplicity makes sense, because transitions are supposed to be executed atomically.
- Variables can be initialized using ordinary assignments and nondeterministic choice statements. The entire initial state may be constrained by a predicate.
- Automaton definitions can be parameterized.
- There is an explicit notation for parallel composition. In order to describe values of variables in the state of a composed automaton, we use a naming convention that prefixes the name of each such state variable with a sequence of names designating the automata of which it is a part. Users can abbreviate some of these names by shorter or more mnemonic "handles." When there is no ambiguity, some of the automaton names or handles in the sequence may be suppressed.
- There are explicit notations for hiding output actions and asserting that a predicate is an invariant of an automaton or that a binary relation is a forward or backward simulation from one automaton to another.

Other languages, such as TLA, UNITY, and SPECTRUM, are similar to IOA in that their basic program units are transition definitions with preconditions and effects. However, effects in TLA are described declaratively, effects in UNITY and SPECTRUM are described imperatively, and we allow both. SPECTRUM also has other features in common with IOA; for example, it uses parameters similar to our `choose` parameters.

The IOA reference manual [GLV97] contains complete definitions of the syntax and semantics of IOA, as well as some sample programs.

13.3.3 A Simple Example: a Communication Channel

The following IOA program presents an abstract view of a reliable FIFO send-receive communication channel. A client can place a message in the channel via a

`send` action, after which the channel can deliver the message via a `receive` action. The specification says nothing about how the channel is implemented to ensure reliable delivery. It simply requires that messages are received in the order they are sent, with no duplicates or omissions.

```
automaton channel(i, j: Node, Node, Msg: type)
  signature
    input send(m: Msg, const i, const j)
    output receive(m: Msg, const i, const j)
  states queue: Seq[M] := {}
  transitions
    input send(m, i, j)
      eff  queue := queue ⊢ m
    output receive(m, i, j)
      pre queue ≠ {} ∧ m = head(queue)
      eff  queue := tail(queue)
```

The automaton is parameterized by two data types, `Node` and `Msg`, which can be instantiated to describe a set of indices for communicating nodes and a set of messages that can be sent; it is also parameterized by the indices `i` and `j` of the sending and receiving nodes. Its signature contains a single `send` and `receive` action for each `m` in `M`; the keyword `const` indicates that the values of `i` and `j` in these actions are fixed by the values of the automaton's parameters. Each channel automaton has a state consisting of a single variable, which holds an initially empty sequence of messages. Its actions are described in terms of their preconditions and effects, using the keywords `pre` and `eff`. An axiomatic definition of the sequence data type provides precise meanings for all other types and operations (`{}` denotes the empty sequence, and $\vdash$ appends an element to a sequence).

IOA can also be used to describe specific implementations for abstract channels, in which lower-level protocols ensure reliable message delivery. Furthermore, it allows a designer to assert that these protocols in fact implement the abstract channel, by defining a relation between the states of the high-level and lower-level automata.

13.4 Extended Example: a Distributed Banking System

In this section, we use IOA to describe a toy banking system in which a single bank account is accessed from several locations, using deposit and withdrawal operations and balance queries. We specify the system and its environment using two I/O automata, `A` and `Env`. `Env` describes what operations can be invoked, where, and when; it represents, for example, a collection of ATMs and customers interacting with those ATMs. Automaton `A` describes what the bank is allowed to do, without any details of the distributed implementation. We also give a formal description `C` of a distributed algorithm that implements `A`, in the context of `Env`. We give a specification `B` for an intermediate service describing stronger guarantees about what the bank does, and we use it to help prove that `C` implements `A` (in the context of `Env`). We also give IOA statements expressing some simple invariants and some

forward simulation relations between the levels. These programs illustrate most of the language constructs.

The code in this section has been checked for validity using our front-end tools, and its correctness has been proved using the Larch Prover.

13.4.1 Banking Environment

The automaton `Env` describes the environment for the banking system. It describes the interface by which the environment interacts with the bank (requests and responses at locations indexed by elements of type `I`), and it expresses "well-formedness conditions" saying that an operation at any location `i` must complete before another operation can be submitted at `i`. `Env` simply keeps track, for each `i`, of whether or not there is an outstanding operation at `i`, and allows submission of a new operation if not.

The definition of `Env` is parameterized by the location type `I`. The output actions of `Env` are requests to perform deposit and withdrawal operations and balance queries. Each request indicates a location `i`. Each deposit or withdrawal request also indicates a (positive) amount `n` being deposited or withdrawn. The `where` predicates are constraints on the action parameters. The input actions of `Env`, which will be synchronized with outputs actions of the bank, are responses `OK(i)` (to deposit and withdrawal requests at location `i`), and `reportBalance(n,i)` (to balance queries).

The only state information is a flag `active[i]` for each location `i`, indicating whether or not there is an active request at location `i`. The rest of the automaton description consists of a collection of TDs that constrain when new requests can be issued. An input at location `i` sets `active[i]` to `false`. An output is allowed to occur at location `i` provided that `active[i]` is `false`, and its effect is to set `active[i]` to `true`. In this description, `Int` and `Bool` are built-in types of IOA, `Array` is a built-in type constructor, and the operator `constant` appearing in the initialization is a built-in operator associated with the `Array` constructor.

```
automaton Env(I: type)
  signature
    input  OK(i: I),
           reportBalance(n: Int, i: I)
    output requestDeposit(n: Int, i: I) where n > 0,
           requestWithdrawal(n: Int, i: I) where n > 0,
           requestBalance(i: I)
  states active: Array[I, Bool] := constant(false)
  transitions
    input OK(i)
      eff  active[i] := false
    input reportBalance(n, i)
      eff  active[i] := false
    output requestDeposit(n, i)
      pre  ¬active[i]
      eff  active[i] := true
    output requestWithdrawal(n, i)
      pre  ¬active[i]
```

```
    eff active[i] := true
  output requestBalance(i)
    pre ¬active[i]
    eff active[i] := true
```

13.4.2 Weak Requirements Specification

Automaton A is an abstract, global description of the basic requirements on the behavior of the banking system. It simply records all deposits and withdrawals in a set of elements of data type OpRec. It allows a balance query to return the result of any set of prior deposits and withdrawals that includes all the operations submitted at the same location as the query. The response need not reflect deposit and withdrawal operations submitted at other locations.

A is parameterized by the location type I. The definition of A introduces several data types: Each OpRec is an "operation record" indicating the amount of a deposit or withdrawal—positive numbers for deposits and negative numbers for withdrawals—plus the location at which it was submitted, a sequence number, and a Boolean value indicating whether the system has reported the completion of the operation to the environment. Each BalRec is a "balance record" indicating the location at which a balance request was submitted and a value to be reported in response. An auxiliary specification Total, written in LSL, defines the function totalAmount, which sums the amount fields in a set of operation records. The type Null[Int] contains a special value null, which indicates the absence of a numerical value. It is used here to indicate that the return value has not yet been determined.

The external signature of A is the "mirror image" of that of Env—its inputs compose with Env's outputs and vice versa. A also has an internal action doBalance, which calculates the balance for a balance query. The state of A consists of four variables: ops holds records of all submitted deposit and withdrawal operations (as OpRecs); bals keeps track of current balance requests (as BalRecs); lastSeqno contains an array of the last sequence numbers assigned to deposits or withdrawals at all locations; and chosenOps is a temporary variable used in one of the TDs.

The functions insert and delete are defined by the built-in data type Set. The program statements involving these functions look slightly complicated because the functions have no side effects. The function nat2pos (defined in the auxiliary specification NumericConversions) converts natural numbers (elements of built-in type Nat) to positive natural numbers (elements of built-in type Pos). The function define converts an element of any type T to an element of type Null[T].

The action requestDeposit causes a new sequence number to be generated and associated with the newly requested deposit operation. The combination of the location at which the operation is submitted and the sequence number serves as an identifier for the operation. The requested deposit amount, the location and sequence number, and the value false indicating that no response for this operation has yet been made to the environment, are all recorded in ops. A

`requestWithdrawal` causes similar effects, only this time the amount recorded is negative. A `requestBalance` causes a record to be made of the balance query, in `bals`.

The action `OK(i)` is allowed to occur any time there is an active deposit or withdrawal operation at location `i`; its effect is to set the `reported` flag for the operation to `true`. The nondeterministic "`choose` parameter" `x` in its TD picks a particular operation record `x` from the set `ops`. The action `doBalance(i)` is allowed to occur any time there is an active balance query at location `i`; its effect is to choose any set of operations that includes all those previously performed at location `i`, to calculate the balance by summing the amounts in all the chosen operations, and to store the result in the balance record in `bals`. Because we are currently using a first-order language, without any special notations for set construction, the effect expresses a set inclusion using an explicit quantifier. Finally, `reportBalance` reports any calculated, unreported balance to the environment.

```
automaton A(I: type)
  type OpRec = tuple of amount: Int, loc: I, seqno: Pos, reported: Bool
  type BalRec = tuple of loc: I, value: Null[Int]
  uses NumericConversions, Total(OpRec, .amount, totalAmount), Null(Int)
  signature
    input requestDeposit(n: Int, i: I) where n > 0,
          requestWithdrawal(n: Int, i: I) where n > 0,
          requestBalance(i: I)
    output OK(i: I),
           reportBalance(n: Int, i: I)
    internal doBalance(i: I)
  states
    ops: Set[OpRec] := {},
    bals: Set[BalRec] := {},
    lastSeqno: Array[I, Nat] := constant(0),
    chosenOps: Set[OpRec]
  transitions
    input requestDeposit(n, i)
      eff lastSeqno[i] := lastSeqno[i] + 1;
          ops := insert([n, i, nat2pos(lastSeqno[i]), false], ops)
    input requestWithdrawal(n, i)
      eff lastSeqno[i] := lastSeqno[i] + 1;
          ops := insert([-n, i, nat2pos(lastSeqno[i]), false], ops)
    input requestBalance(i)
      eff bals := insert([i, null], bals)
    output OK(i)
      choose x: OpRec
      pre x ∈ ops ∧ x.loc = i ∧ ¬x.reported
      eff ops := insert(set_reported(x, true), delete(x, ops))
    output reportBalance(n, i)
      pre [i, define(n)] ∈ bals
      eff bals := delete([i, define(n)], bals)
    internal doBalance(i)
      pre [i, null] ∈ bals
      eff chosenOps := choose c
            where ∀ y:OpRec (y.loc = i ∧ y ∈ ops ⇒ y ∈ c) ∧ c ⊆ ops;
          bals := insert([i,define(totalAmount(chosenOps))],delete([i, null], bals))
```

Automaton `AEnv` is the parallel composition of automata `A` and `Env`, matching external actions:

```
automaton AEnv(I: type)
  compose A(I); Env(I)
```

The programmer can state invariants of `AEnv` within IOA. In the following invariant, the first clause implies that the value of the variable `lastSeqno[i]` is greater than or equal to all sequence numbers that have ever been assigned to operations originating at location `i`. The second clause implies that the sequence numbers assigned to operations submitted at location `i` form a prefix of the positive integers. The third and fourth clauses say that the environment's `active[i]` flag correctly indicates when an operation or balance query is active, and also say that only one operation is active at any location at any time. The final clause says that the location and sequence number together identify an operation in `ops` uniquely.

```
invariant of AEnv:
    ∀ x:OpRec (x ∈ ops ⇒ pos2nat(x.seqno) ≤ lastSeqno[x.loc])
  ∧ ∀ i:I ∀ k:Pos (pos2nat(k) ≤ lastSeqno[i]
                      ⇒ ∃ z:OpRec (z ∈ ops ∧ z.loc = i ∧ z.seqno = k))
  ∧ ∀ x:OpRec
       (   x ∈ ops ∧ ¬x.reported
        ⇒    active[x.loc]
           ∧ ∀ y:OpRec (y ∈ ops ∧ x.loc = y.loc ∧ ¬y.reported ⇒ x = y)
           ∧ ∀ b:BalRec (b ∈ bals ⇒ x.loc ≠ b.loc))
  ∧ ∀ b:BalRec
       (b ∈ bals ⇒   active[b.loc]
                   ∧ ∀ b1:BalRec (b1 ∈ bals ∧ b.loc = b1.loc ⇒ b = b1))
  ∧ ∀ x:OpRec ∀ y:OpRec
       (x ∈ ops ∧ y ∈ ops ∧ x.loc = y.loc ∧ x.seqno = y.seqno ⇒ x = y)
```

13.4.3 Strong Requirements Specification

Automaton `B` is very much like `A`, but imposes a stronger requirement, namely, that the response to a balance query include the results of all deposits and withdrawals anywhere in the system that complete before the query is issued. It does this by adding a state variable `mustInclude[i]` of type `Array[I, Set[OpRec]]` to `A`, by appending the statement

```
mustInclude[i] := choose s where ∀ x:OpRec (x ∈ s ⇔ x ∈ ops ∧ x.reported)
```

to the effect of the `requestBalance(i)` TD, and by modifying the `choose` statement in the `doBalance(i)` TD to require the chosen set `c` of operations to include `mustInclude[i]`. The changed parts appear below.

```
automaton B(I: type)
  ...
  states
    ...
    mustInclude: Array[I, Set[OpRec]] := constant({})
  transitions
    input requestBalance(i)
      eff  bals := insert([i, null], bals);
           mustInclude[i] := choose s where
              ∀ x:OpRec (x ∈ s ⇔ x ∈ ops ∧ x.reported)
    internal doBalance(i)
      pre [i, null] ∈ bals
      eff  chosenOps := choose c where
```

```
        ∀ y:OpRec (y.loc = i ∧ y ∈ ops ⇒ y ∈ c)
        ∧ mustInclude[i] ⊆ c ∧ c ⊆ ops;
    bals := insert([i, define(totalAmount(chosenOps))],delete([i,null], bals))

automaton BEnv(I: type)
  compose B(I); Env(I)
```

Informally, it is easy to see that `BEnv` implements `AEnv` in the sense that every trace of `BEnv` is also a trace of `AEnv`. Formally, this can be shown using a trivial forward simulation relation from `BEnv` to `AEnv`, namely, the identity relation for the state variables of `AEnv`. This relation can be expressed in IOA as follows, using our prefix naming convention for variables in a composition. Since there is no ambiguity, we can write, for example, `AEnv.active` and `A.ops` as abbreviations for the complete names `AEnv.A.active` and `AEnv.A.ops`, respectively.

```
forward simulation from BEnv to AEnv:
    AEnv.active = BEnv.active ∧ A.ops = B.ops ∧ A.bals = B.bals
  ∧ A.lastSeqno = B.lastSeqno ∧ A.chosenOps = B.chosenOps
```

13.4.4 Distributed Implementation

Now we describe a distributed implementation as an automaton `C` that is the composition of a node automaton `CO(i)` for each `i` in `I`, plus reliable FIFO send/receive communication channels `channel(i,j)` for each pair of distinct `i` and `j` in `I`, as described in Section 13.3.3. Each node automaton `CO(i)` keeps track of the set of deposit and withdrawal operations that it "knows about," including all the local ones. It works locally to process deposits and withdrawals, but a balance query causes it to send explicit messages to all other nodes. It collects responses to these messages and combines them with its own known operations to calculate the response to the balance query.

Since the automaton `CO(i)` corresponds to a location `i`, its action names are parameterized by `i`. Its `send` and `receive` actions are intended to match the same-named channel actions. In the state of `CO(i)`, `ops` is maintained as a set of records with no `reported` field; each record is an element of a new type `OpRec1`. The information about which operations have been completed is kept locally in a separate variable `reports`, and is not sent in messages. Balance information is also recorded locally, as elements of a new type `BalRec1`, and never sent. Additional state variables keep track of request messages that have been sent, response messages that have been received, and response messages that must be sent. Specifically, the Boolean flag `reqSent[j]` is used to keep track of whether a `req` message has been sent to `j`, and the Boolean flag `respRcvd[j]` is used to keep track of whether a response has been received from `j`. The flag `reqRcvd[j]` is used to record that a request has just been received from `j` and is waiting to be answered. (Although these flag arrays are indexed by all of `I`, the flags for `i` itself are not really needed.) Since two kinds of messages are sent in this algorithm, we define a new message type `Msg` as the union of the two individual types.

```
automaton C0(i: I, I: type)
  type OpRec1 = tuple of amount: Int, loc: I, seqno: Pos
  type BalRec1 = tuple of value: Null[Int]
  type Msg = union of set: Set[OpRec1], req: String
  uses NumericConversions, Total(OpRec1, .amount, totalAmount), Null(Int)

  signature
    input requestDeposit(n: Int, const i) where n > 0,
          requestWithdrawal(n: Int, const i) where n > 0,
          requestBalance(const i),
          receive(m: Msg, j: I, const i) where j ≠ i
    output OK(const i),
           reportBalance(n: Int, const i),
           send(m: Msg, const i, j: I) where j ≠ i
    internal doBalance(const i)

  states
    ops: Set[OpRec1] := {},
    reports: Set[Pos] := {},
    bals: Set[BalRec1] := {},
    lastSeqno: Nat := 0,
    reqSent: Array[I, Bool] := constant(false),
    respRcvd: Array[I, Bool] := constant(false),
    reqRcvd: Array[I, Bool] := constant(false)

  transitions
    input requestDeposit(n, i)
      eff lastSeqno := lastSeqno + 1;
          ops := insert([n, i, nat2pos(lastSeqno)], ops)
    input requestWithdrawal(n, i)
      eff lastSeqno := lastSeqno + 1;
          ops := insert([-n, i, nat2pos(lastSeqno)], ops)
    input requestBalance(i)
      eff bals := insert([null], bals);
          reqSent := constant(false);
          respRcvd := constant(false)
    output OK(i)
      choose x: OpRec1
      pre x ∈ ops ∧ x.loc = i ∧ ¬((x.seqno) ∈ reports)
      eff reports := insert(x.seqno, reports)
    output reportBalance(n, i)
      pre [define(n)] ∈ bals
      eff bals := delete([define(n)], bals)
    internal doBalance(i)
      pre [null] ∈ bals ∧ ∀ j:I (j ≠ i ⇒ respRcvd[j])
      eff bals := insert([define(totalAmount(ops))], delete([null], bals))
    output send(req(x), i, j)
      pre ¬reqSent[j] ∧ [null] ∈ bals
      eff reqSent[j] := true
    output send(set(m), i, j)
      pre m = ops ∧ reqRcvd[j]
      eff reqRcvd[j] := false
    input receive(set(m), j, i)
      eff ops := ops ∪ m;
          respRcvd[j] := true
    input receive(req(x), j, i)
      eff reqRcvd[j] := true
```

We define `C` to be the composition of all the `C0(i)` and all the channels, with the communication actions hidden (to match the external signature of `B`), and `CEnv` to be the composition of `C` with the environment.

```
automaton C(I: type)
  compose C0(i) for i: I; channel(i, j, I, Msg) for i: I, j: I where i ≠ j
  hide send(m, i, j), receive(m, i, j) for m: Msg, i: I, j: I
```

```
automaton CEnv(I: type)
  compose C(I); Env(I)
```

CEnv has invariants analogous to those of AEnv, as well as trivial invariants saying that channels from nodes to themselves are never used. A new invariant says that any (deposit or withdrawal) operation that appears anywhere in the state (at a node or in a message) also appears in ops at its originating location. Other invariants express consistency conditions such as the following. (a) If there is a request in a channel, then there is an active query, the flags for sending and receiving are set correctly, there is only one request in that channel, and there is no response in the return channel. (These last two conclusions rule out messages left over from earlier balance queries.) (b) If there is a response in a channel, then there is an active query, the flags are set correctly, there is only one response in the channel, and there is no request in the corresponding channel. (c) The sending and receiving flags are set consistently. (d) If a response has been received, then a corresponding request was sent. We omit the IOA formulations of these invariants here; the technical report [GL98] contains them all.

To show that CEnv implements BEnv, we define a forward simulation relation from CEnv to BEnv. This uses a projection function proj from OpRecs to OpRec1s, defined in an auxiliary specification Projections, that just eliminates the reported component.

```
uses Projection
forward simulation from CEnv to BEnv:
    BEnv.active = CEnv.active
  ∧ ∀ x:OpRec
        (x ∈ B.ops ⇔    proj(x) ∈ C0(x.loc).ops
                      ∧ (x.reported ⇔ x.seqno ∈ C0(x.loc).reports))
  ∧ ∀ x:BalRec (x ∈ B.bals ⇔ [x.value] ∈ C0(x.loc).bals)
  ∧ ∀ i:I (B.lastSeqno[i] = C0(i).lastSeqno)
  ∧ ∀ i:I ∀ j:I ∀ x:OpRec
        (   [i,null] ∈ B.bals ∧ x ∈ B.mustInclude[i] ∧ x.loc = j ∧ j ≠ i
         ⇒    proj(x) ∈ C0(j).ops
           ∧ ∀ m:Set[OpRec1](set(m) ∈ channel(j,i).queue ⇒ proj(x) ∈ m)
           ∧ (C0(i).respRcvd[j] ⇒ proj(x) ∈ C0(i).ops))
```

The first four conjuncts define simple correspondences between the ops, bals, lastSeqno, and active components in BEnv and CEnv. The last conjunct says that, if there is an active balance query at location i, and if operation x, originating at another location j, is one of those that must be included in the query, then x must appear in certain places in the global state of CEnv. In particular, x must be in ops at location j, must be in any response message in transit from j to i, and, in case i has received a message from j, must be at location i. The existence of this forward simulation implies that CEnv implements BEnv, which in turn implies that CEnv implements AEnv.

13.5 Discussion of Language Design

Nondeterminism is an important feature of IOA, because it allows programmers to avoid restricting their designs unnecessarily. Reasoning about a design in a general form is desirable because it produces insights (and theorems) that may apply to many different implementations. Removing the "clutter" of unnecessary restrictions makes it easier to understand why designs work, because it is easier to see what correctness properties really depend on.

An important aspect of nondeterministic programming is allowing maximum freedom in the order of action execution. In specifications for interactive programs, considerable freedom in action order is often acceptable. Unlike traditional sequential programming styles, the guarded command style used in IOA makes it easy for programmers to constrain action order only when necessary.

Of course, control over action order is sometimes needed, particularly at lower levels of abstraction where performance requirements may force particular scheduling decisions. The current version of IOA lacks explicit control structures for describing such constraints. (In examples, these have generally been expressed using special *pc* or *status* variables to track progress in the sequential part of a computation.) It is likely that we will want later to enhance IOA with explicit support for specifying action order.

However, new research is needed to discover how best to do this. Standard sequential control constructs are neither sufficient nor entirely necessary. For example, reactive systems may contain threads that are intended to execute sequentially, but can be interrupted at any time; describing interactions between threads and interrupt-handling routines may require special control structures. On the other hand, guarded commands can be used to describe iteration, which suggests that some standard looping constructs can be avoided. In any case, to maintain simplicity and provability and to ensure consistency with the mathematical model, we think that new sequencing constructs should be added as pure *syntactic sugar*, that is, that there should be an unambiguous translation of the code with the additions into code without them.

Another possible improvement to IOA would add further local naming conventions. For instance, currently all of an automaton's state variables are global to all of its TDs; one could add variables whose scope is limited to a single TD. Also, currently all action names in a composition are global. One could also allow local action names, with a more flexible method of matching up names in a composition; SPECTRUM [Gol90] uses such a mechanism. A renaming operator for actions would also be useful.

It might also be desirable to add other "standard" programming language features to IOA. (The addition of some object-oriented features to I/O automata is described in [BH98].) However, we think that such features should be added judiciously,

to avoid complicating the semantics of IOA. In particular, we think that such extensions should be made as syntactic sugar.

Similarly, it might be desirable to enrich the logical and mathematical features of IOA. We have chosen to base IOA on LSL, which uses the familiar syntax and semantics of first-order logic, so as to facilitate translation into the input languages of several different theorem provers. As a result, we must rewrite an informal statement such as $s := \{n : n < 10 \wedge a[n] > 0\}$ as an IOA statement

```
s := choose x where ∀ n: Int (n ∈ x ⇔ n < 10 ∧ a[n] > 0)
```

that uses explicit quantifiers. Although theorem provers such as PVS and Isabelle provide richer notations than LP, we are not attracted to gaining expressive power by tying IOA too closely to less widely understood notations and type systems, which might limit the range of tools with which IOA could be employed. Instead, we envision two ways of gaining expressive power. One is to enrich IOA with syntactic sugar for particularly useful constructs. Another is to base IOA on the new Common Algebraic Specification Language (CASL) [CoF98] and leverage the work of others in translating CASL specifications into the input languages of different theorem provers. CASL is attractive because it is an emerging standard, has a richer type system than LSL, and provides better support for parameterized specifications.

A different, and more traditional, approach to constructing verified code has been to begin with a rich, expressive programming language, define formal semantics and proof rules, and try to use them for verification. We think that this approach has a serious problem: complicated languages have complicated semantics and complicated proof rules, which are difficult to think about and difficult to manipulate in proofs. The logical complexity of a design described in such a language becomes intertwined with the complexity of the language, making it hard to understand and verify the design. We think that a better approach is to begin with a very simple language that supports good proofs for high-level designs, and to add constructs carefully to obtain expressiveness.

13.6 Tools

In this section, we describe a set of tools to support IOA programming, and we describe our progress in building prototypes. For uniformity of presentation, we describe all tools in the present tense, although they are actually in various stages of development (as indicated at the end of each tool's description).

13.6.1 General Guidelines

We require that all tools be based formally on the mathematical model. The tools should be accompanied by theory to explain their operation, for example, theorems about the correctness of program transformations and theorems about the correctness and performance of generated code.

Not all tools need to be capable of processing the full IOA language. Some tools may only process restricted forms of programs, with the user responsible for transforming programs into the restricted forms.† This approach allows users to express their designs in the general IOA language, yet still utilize tools, like simulators and model checkers, that require restrictions. In particular, we believe that the user should help resolve scheduling decisions and other forms of nondeterministic choice when submitting a program to a simulator or code generator.

The most important use of the validation tools will be for checking safety properties. In fact, we propose de-emphasizing liveness properties in favor of considering *time bounds*, which yield sharper information, can be expressed formally as safety properties, and can be handled using standard assertional methods.†

The entire toolset, except for the theorem prover, should be usable by skilled programmers. Use of the theorem prover will require a fair amount of skill in logic and formal methods.

13.6.2 Basic Support Tools

The basic tools for IOA include a *front end*, consisting of a parser and static semantic checker, which produces an internal representation suitable for use by the other (*back-end*) tools. Other basic tools support structured system descriptions using composition and levels of abstraction.

To support composition, a *composer* tool converts the description of a composite automaton into *primitive form* by explicitly representing its actions, states, transitions, and tasks. The input to the composer must be a *compatible* collection of automata, which means, for example, that the component automata must have no common output actions. (This compatibility can be verified using other tools—in simple cases, the static semantic checker, and in more complicated cases, a theorem prover.) In the resulting automaton description, the name of a state variable is prefixed with the names (or handles) of the components from which it arises.

To support levels of abstraction, the tools provide facilities for defining and using *simulation relations*. When users argue that one automaton A implements another automaton B, they normally expect to supply a predicate relating the states of A and B. We think it is reasonable to expect the user to supply more, in particular, information relating *steps* of the two automata. Such information can be used by a theorem prover in establishing the correctness of a simulation relation (see Section 13.6.3), or by a simulator in testing its correctness (see Section 13.6.4).

For example, to show that a relation R is a forward simulation from A to B, the user can define, for each step (s_A, π, s'_A) of A (arising from a given TD), and for each state s_B of B such that $(s_A, s_B) \in R$, a "corresponding" execution fragment

† We use "user" to denote the user of the toolset, that is, the system designer, programmer, or program validator.

† Incorporating time bounds formally into IOA requires an extension to timed I/O automata, which is beyond the scope of this paper.

of B. One way he/she can specify this fragment is by providing, as a function of the given step and state, (a) a sequence of TDs of B, and (b) a way of resolving the explicit nondeterministic choices (those represented by `choose` statements and parameters) in those TDs. This function can be described using cases, based on a user-defined classification of the steps of A. To resolve nondeterministic choices, the user can supply subroutines. The programming environment provides an API for use in defining such step correspondences.

It is not always clear how to define the needed execution fragment solely as a function of the given step and state. For example, the definition of the fragment might depend on explicit nondeterministic choices or on the outcomes of conditional tests in the step of A. In such cases, the user can add *history variables* to A to record the relevant choices, and use the values of these variables in state s'_A in defining the fragment. The tools support the addition of such history variables.

We have implemented the front end already. Chefter's Master's Thesis [Che98] describes a design for the composer. Neither the composer nor support for levels of abstraction has been implemented yet.

13.6.3 Interfaces to Proof Tools

The toolset includes interfaces to existing theorem provers. The IOA language was designed for easy translation into axioms that can be used by interactive theorem provers. In this translation, all imperative statements in the effects of TDs, including assignment statements, `choose` statements, conditionals, and loops, are replaced by predicates relating poststates to prestates, and similarly for initial state descriptions. Other axioms are derived from formal definitions of the data types used in the automata.

Theorem provers can be used to prove validity properties for IOA programs and other user inputs (for example, that the set of choices for a nondeterministic assignment is nonempty, that automata being composed do not share output actions, or that actions specified by the user of the simulator are enabled) in cases where the properties are too hard to establish by static checking. Theorem provers can also be used to prove properties of data types used in automata, invariants of automata, and simulation relations between automata. Theorem provers must be able to process programs written in the full IOA language.

For example, showing that a relation R is a forward simulation from A to B involves showing a relationship between the start states of A and B and a relationship between the steps of A and B. The latter asserts, for each step of A and each state of B that is R-related to the pre-state in A, the existence of a "corresponding" fragment of B. Proving such an existence statement automatically is difficult for theorem provers, so the interface can ask the user to help by supplying explicit step correspondence information, as described in Section 13.6.2. The user can then use the theorem prover to verify that the specified sequence satisfies the requirements

for a forward simulation: that the sequence is really an execution fragment, that it has the same external behavior as the given step, and that the final states are related by R. Our experience with proofs of distributed algorithms indicates that such step correspondence information greatly reduces the amount of interaction needed for the theorem prover to complete its work.

Initially, we are developing an interface to the Larch Prover. We have designed a translation scheme from IOA descriptions into LSL and used it manually to prove the invariants and simulation relations shown in Section 13.4 [GL98]. We have formalized the translation scheme (cf. [GLV97]) and are in the process of implementing it. Meanwhile, Devillers, working with Vaandrager, is writing a translation from IOA descriptions to the input language of PVS [Dev99].

The toolset also includes interfaces to existing model checkers. These interfaces only handle a restricted class of IOA programs. Although programs can be nondeterministic, they must be written in an imperative style and use only those data types provided by the model checker's input language.

So far, Vaziri has written a preliminary translation from a restricted class of IOA programs into PROMELA, the input language of the SPIN model checker [Hol91].

13.6.4 Simulator

The simulator runs sample executions of an IOA program on a single machine, allowing the user to help select the executions. The simulator is used mainly for checking proposed invariants and simulation relations.†

The simulator requires that IOA programs be transformed into a restricted form. The biggest problem in this transformation is resolving nondeterminism, which appears in IOA in two ways: *explicitly*, in the form of `choose` constructs in state variable initializations and TD effects, and *implicitly*, in the form of action scheduling uncertainty. The restricted form rules out both types of nondeterminism. We also assume that an IOA program submitted to the simulator is *closed* (that is, has no input actions) and is written in an imperative style. At most one (locally controlled) action may be enabled in any state; moreover, the user is expected to designate that action, as a function of the current state.

The tools provide support for getting programs into the required form. For example, the composer can be used to "close" an automaton by composing it with a user-defined "environment automaton" (like `Env` in Section 13.4.1). To resolve explicit nondeterminism, the system can generate probabilistic choices. Alternatively, the system can ask the user to provide explicit choices, for example, by adding a state variable containing a pseudo-random sequence and replacing nondeterministic choices by successive elements of this sequence. The theorem prover can be used to check that the provided choices satisfy any required constraints, expressed by `where` clauses, preconditions, and other predicates.

† There is an unfortunate clash of terminology here, between "simulator" and "simulation relation."

To remove implicit nondeterminism, the system can ask the user to constrain the automaton so only one action is enabled in each state; the user can do this, for example, by adding state variables containing scheduling information, adding extra preconditions for actions that involve the new variables, and adding new statements to the effects of actions to maintain the scheduling variables. The user should also provide a function that explicitly designates the next action to be simulated, as a function of the current state, in the form of a TD plus expressions giving values for the action's parameters. The programming environment provides an API for use in writing these functions, and the theorem prover can be used to verify that the designated action is enabled.

For example, if a set of actions is to be executed in round-robin order, then a scheduling variable can keep track of the (index of the) next action to be performed, the precondition of each action can be augmented with a clause saying that the indicated action is the one recorded by this variable, and the effect of each action can increment the index maintained by the variable. This strategy removes the scheduling nondeterminism, and an explicit function of the state describes the next action to be performed.

In order to simulate data type operations, which are defined axiomatically in IOA, the simulator needs actual code. For operations defined by IOA's built-in data types, the simulator uses code from class libraries written in a standard sequential programming language like C++ or Java. For operations (like `totalAmount` in Section 13.4) defined in auxiliary LSL specifications, the user can choose either to write these specifications in an executable algebraic style or to supply handwritten code. Although we do not plan to prove the correctness of this handwritten code, such proofs could be carried out using techniques of sequential program verification.

With all nondeterminism removed, the simulator's job is easy: starting from the unique initial state, it repeatedly performs the unique enabled action. That is, it uses the user-provided function to determine the next TD and parameter values, then executes that TD with those parameter values. Since there is no explicit nondeterminism, this uniquely determines the next state.

The simulator can be used to check that proposed invariants are true in all states that arise in the simulated executions. It can also check that a candidate relation R appears to be a simulation relation from A to B by performing a *paired simulation* of A and B, that is, by producing an execution of A as usual and using it to generate a corresponding execution of B. Specifically, for each simulated step of A, the simulator uses a user-specified step correspondence (see Section 13.6.2) to obtain a (proposed) execution fragment of B, then runs the steps of that execution fragment. As it runs those steps, the simulator checks that action preconditions are satisfied, that values used to resolve explicit nondeterministic choices satisfy the required constraints, that the fragment has the same external behavior as the given step, and that the relation R holds between the states of the two automata after the step and fragment.

Chefter's Master's Thesis [Che98] contains a detailed design for the basic simulator; she has also written a preliminary implementation in Java and produced a small library of hand-coded data type implementations. More recently, Ramirez has improved the simulator, this time starting from the intermediate language described in Section 13.6.2. It remains to enhance this newer version with more advanced capabilities, including the resolution of explicit and implicit nondeterministic choices and support for paired simulations.

13.6.5 Code Generator

Nearly all of the issues that arise in the simulator arise also for the code generator. New issues also arise because of distribution, the need to interact with externally provided communication services, and the need for good runtime performance.

The code generator generates real code for a target distributed system, which may be an arbitrary configuration of *computing nodes* and *communication channels.* The code generation scheme works directly from a low-level IOA language description of the system design, which can arise from a series of refinements starting with a high-level specification. This strategy allows the formal modeling and analysis facilities to be used to reason about the design until the last possible moment, when it is transformed automatically into a working implementation. The verification facilities can be used to ensure that the final implementation provably implements higher-level IOA descriptions, subject to assumed properties of externally provided services, of hand-coded data type implementations, and of the underlying hardware.

The code generation scheme produces runnable versions of *node automata* that can communicate via preexisting communication services such as TCP or MPI [MPI95], which are modeled by *channel automata.* Node automata typically model a combination of application-specific code and local pieces of communication protocols. A key to making this scheme work is obtaining clear IOA specifications of real communication services. Such definitions may be obtained by formalizing existing informal interface descriptions and recasting them, if necessary, in terms of shared actions.

The code generator, like the simulator, relies on users to transform programs into a special form. As just described, programs provided as input to the code generator must match the given distributed system architecture. Node programs must also satisfy restrictions like those required by the simulator, although they need not be closed. That is, they should include neither explicit nor implicit nondeterminism. As before, users should specify the next enabled action, as a function of the state.

We need another (technical) restriction to get a faithful system implementation. Atomicity requires that the effect of each transition occur without interruption, even if inputs arrive from clients or communication services during its execution. In our design, such inputs are buffered. In between running locally controlled actions, the generated program examines buffers for newly arrived inputs, and handles some

or all of them by running code for input actions. Since this delays processing inputs (with respect to when the corresponding outputs occur), it may upset precise implementation claims for the node automata. Therefore, we restrict node programs in order to avoid this risk. Stated in its strongest, simplest form, our restriction is that each node automaton A be *input-delay-insensitive*: its external behavior should not change if its input actions are delayed and reordered before processing.† It is possible to weaken this requirement slightly, for example, requiring only that external behavior be preserved in "well-formed" environments, for example, only for blocking inputs. In this case, the actual environments of the node automata must satisfy these assumptions.

As for the simulator, the tools provide help in getting programs into the special form for the code generator. The general tools that support programming using levels of abstraction can be used to refine a design within the IOA framework until the required node-and-channel form is reached. IOA specifications for actual communication services like TCP and MPI are maintained in a library. Support for removing explicit (`choose`) and implicit (scheduling) nondeterminism is similar to that for the simulator. Like the simulator, the code generator uses a library of data type implementations.

For each node automaton, the code generator performs a source-to-source translation, translating the IOA code into a program in a standard programming language like C++ or Java. This program performs a simple loop, similar to the one performed by the simulator, except that it polls and handles input actions in between processing locally controlled actions. The code generator may translate the code at different nodes into different programming languages.

By insisting that IOA programs from which we generate code match the available computing hardware and communication services, and by requiring the node programs to tolerate input delays, we can achieve a faithful implementation without using any nonlocal synchronization, such as that required by earlier designs [Gol91, Che97].

Abstract Channels Before using the code generator, it is often helpful to describe a system design as a composition of application automata A_i and high-level *abstract channel automata* C_{ij}. Each C_{ij} is, in turn, implemented by lower-level automata D_{ij} and D_{ji}, representing real channels, composed with protocol automata P_{ij} and P_{ji}. For example, an abstract FIFO send/receive channel can be implemented in terms of an MPI service and an IOA protocol [Tau]. At this lower level of design, a node automaton N_i is, formally, the composition of the application automaton A_i and all the protocol automata P_{ij} (for node i) that appear in the channel implementations. It is this composed automaton N_i that the code generator translates

† Formally, $traces(A' \times Buff)$ must be a subset of $traces(A)$, where A' is like A except that its inputs are renamed to internal versions, and *Buff* is a possibly-reordering delay buffer that takes the real inputs and delivers them later in their internal versions.

into a standard programming language. Figure 13.1 illustrates this design; in this figure, the composed automata N_i are encircled by dotted lines.

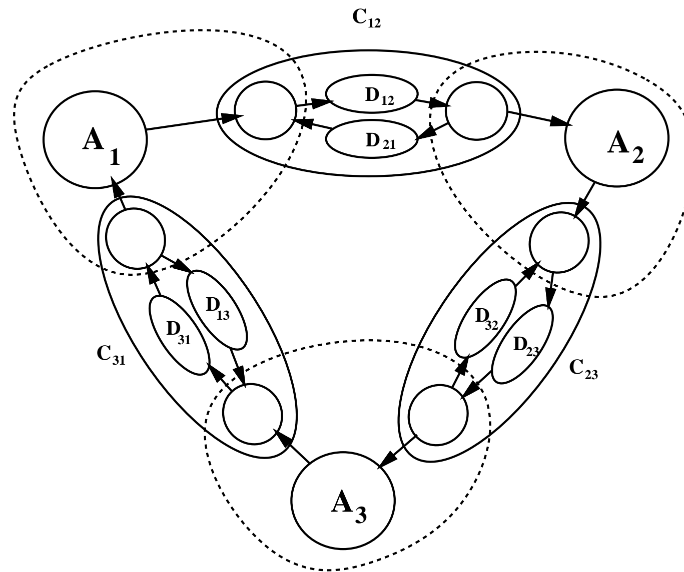

Fig. 13.1. An Implementation Using Abstract Channels Implemented by Real Channels.

Abstract channels provide flexibility: different abstract channels can be used with the same distributed system architecture, and the same abstract channels can be used with different architectures. The tools support programming with abstract channels by maintaining libraries of IOA descriptions of abstract and real channels, and libraries of IOA implementations of abstract channels in terms of real channels. The tools can also assist in proving the correctness of implementations of abstract channels.

Status of Code Generation Tauber has defined IOA models for external system services, including the console and a subset of MPI functions. In a first project on abstract channels, he has defined a protocol that implements reliable FIFO channels on top of MPI and proved the correctness of this protocol [Tau]. He has hand-translated sample distributed IOA programs using abstract channels into Java, and he has built an initial version of the code generator for a restricted subset of IOA, using MPI with a Java wrapper [BCF$^+$99]. This initial version resolves action-scheduling nondeterminism using a simple round-robin scheduler.

Tauber and Tsai, with help from Ramirez and Reimers, are currently working on a reimplementation of the code generator. This version takes the IOA intermediate language as input instead of the source code, uses a flexible action scheduler, and

does not restrict the use of action parameters, as does the initial version. The new version is constructed as a series of program transformations that use the composer and remove various kinds of nondeterminism.

13.7 Conclusions

It must still be shown that distributed code with acceptable performance can be obtained using IOA. We have many reasons for believing it can. First and foremost, our strategy works *locally*, without synchronizing any activities involving more than one machine. Also, the code-generation process incorporates existing services (for example, communication services), which may be highly optimized. Also, we allow hand coding of data type implementations in a standard sequential programming language, which provides many opportunities for optimization.

We think that giving the user flexibility in controlling the order in which actions of a local node program are performed will yield more efficient schedules than would arise from having a fixed scheduling discipline. Allowing the scheduler to call a user-provided function to determine the next action should decrease runtime overhead. Proving some properties statically should save the expense of some runtime checks.

Source-to-source translation to C++ or other languages allows the use of optimizing compilers for those languages. Also, IOA is sufficiently flexible to be used at different levels of abstraction, including a very low level that can permit detailed optimization within IOA itself.

Many research problems remain. For theorem prover support, an interesting problem is to devise specialized proof strategies for proving invariants, simulation relations, and IOA program validity properties, in order to reduce the amount of interaction needed in proofs. For model checker support, it would be useful to augment existing model checkers with additional data types so they can express a larger class of IOA programs. Also, one could develop support for exploring restricted subsets of an automaton's execution (for example, based on limiting the amount of asynchrony) in situations where the full automaton is too large to model check. Another interesting problem is to develop support for model checking proposed simulation relations, based on the notion of *paired simulation* described in Section 13.6.4. For the simulator, it would be useful to improve the support for resolving implicit nondeterminism by developing a library of built-in schedulers, and to develop an API to help users construct new schedulers.

For the code generator, research will be needed on improving performance and usability. For example, the target code for node programs could use multithreading to improve performance; however, this would make atomicity of actions harder to ensure and introduce concurrency control issues. Additional support for the user in resolving implicit and explicit nondeterminism could be developed. A more sophisticated, easier-to-use scheduling facility could be developed and integrated into the toolset.

Users can program in IOA at any level of abstraction. Low levels include more aspects of program behavior and so permit finer-tuned optimization. An interesting general problem is to determine how high a level of abstraction programmers can use and still obtain acceptable implementation performance.

Finally, research is needed in using these methods to generate prototype application systems, in evaluating these systems, in using the results of these experiments to improve the code generation methods, and in using the improved methods to develop useful distributed applications.

Acknowledgments

We thank Anna Chefter, Antonio Ramirez, Joshua Tauber, and Mandana Vaziri for their many contributions to this project, especially Josh for his help with Section 13.6.5. We also thank Michael Tsai and Holly Reimers for their programming assistance, and Jason Hickey, Martin Rinard, and Frits Vaandrager for their suggestions and encouragement. Garland's work was supported in part by NSF Grant CCR-9504248. Lynch's work was supported in part by DARPA contract F19628-95-C-0018 (monitored by Hanscom Air Force Base), AFOSR contract F49620-97-1-0337, NSF grants CCR-9225124 and CCR-9804665, and NTT Proposal MIT9904-12.

Bibliography

[AHS98] Archer, M. M., Heitmeyer, C. L., and Sims, S. TAME: A PVS interface to simplify proofs for automata models. In *Workshop on User Interfaces for Theorem Provers*, Eindhoven University of Technology, July 1998.

[Arc97] Archer, M., 1997. Personal communication.

[BCF+99] Baker, M., Carpenter, B., Fox, G., Ko, S. H., and Lim, S. mpiJava: An object-oriented Java interface to MPI. In *Int. Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, San Juan, Puerto Rico, April 1999.

[BH98] Bickford, M. and Hickey, J. Composition and inheritance for I/O automata using intersection types. Dept. of Computer Science, Cornell University, 1998.

[BKT92] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. Orca: A language for parallel programming of distributed systems. *IEEE Trans. on Soft Eng.*, 18(3):190–205, March 1992.

[Che97] Cheiner, O. Implementation and evaluation of an eventually-serializable data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1997.

[Che98] Chefter, A. E. A simulator for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1998.

[CM88] Chandy, K. M. and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Co., Reading, MA, 1988.

[CoF98] The common algebraic specification language, April 1998. `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

[Dev99] Devillers, M. Translating IOA automata to PVS. Preliminary Research Report CSI-R9903, Computing Science Institute, University of Nijmegen, the Netherlands, feb 1999.

[DFLS98] DePrisco, R., Fekete, A., Lynch, N., and Shvartsman, A. A dynamic view-oriented group communication service. In *Proc. 17th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 227–236, Puerto Vallarta, Mexico, June–July 1998.

[FGL+99] Fekete, A., Gupta, D., Luchangco, V., Lynch, N., and Shvartsman, A. Eventually-serializable data service. *Theoretical Computer Science*, 220(1):113–156, June 1999.

[FKL98] Fekete, A., Kaashoek, M. F., and Lynch, N. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1):35–69, January 1998.

[FLS97] Fekete, A., Lynch, N., and Shvartsman, A. Specifying and using a partitionable group communication service. In *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 53–62, Santa Barbara, CA, August 1997. Expanded version in Technical Memo MIT-LCS-TM-570, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1997.

[Gar94] Garland, S. J. LP, the Larch Prover, version 3.1, December 1994. MIT Laboratory for Computer Science. `http://www.sds.lcs.mit.edu/~garland/LP/overview.html`.

[GG91] Garland, S. J. and Guttag, J. V. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[GH93] Guttag, J. V. and Horning, J. J., editors. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag Texts and Monographs in Computer Science, 1993. With S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing.

[GL98] Garland, S. J. and Lynch, N. A. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998. `http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[GLV97] Garland, S. J., Lynch, N. A., and Vaziri, M. *IOA: A Language for Specifying, Programming and Validating Distributed Systems*. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 1997. `http://sds.lcs.mit.edu/~garland/ioaLanguage.html`.

[Gol90] Goldman, K. J. *Distributed Algorithm Simulation using Input/Output Automata*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1990.

[Gol91] Goldman, K. J. Highly concurrent logically synchronous multicast. *Distributed Computing*, 6(4):189–207, 1991.

[HLvR99] Hickey, J., Lynch, N., and van Renesse, R. Specifications and proofs for ENSEMBLE layers. In Cleaveland, R., editor, *Tools and Algorithms for the Construction and Analysis of Systems* (Fifth International Conference, TACAS'99), volume 1579 of *Lecture Notes in Computer Science*, pages 119–133, Amsterdam, the Netherlands, March 1999. Springer-Verlag.

[Hoa85] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall International, United Kingdom, 1985.

[Hol91] Holzmann, G. J. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, New Jersey, 1991.

[HvR96] Hayden, M. and van Renesse, R. Optimizing layered communication protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell

University, Ithaca, NY, November 1996.

[Lam94] Lamport, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LMWF94] Lynch, N., Merritt, M., Weihl, W., and Fekete, A. *Atomic Transactions*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[LT87] Lynch, N. A. and Tuttle, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, British Columbia, Canada, August 1987.

[LV95] Lynch, N. and Vaandrager, F. Forward and backward simulations—Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[LV96] Lynch, N. and Vaandrager, F. Forward and backward simulations—Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[Lyn96] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[Mil89] Milner, R. *Communication and Concurrency*. Prentice-Hall International, United Kingdom, 1989.

[MPI95] MPI: A message-passing interface standard,Version 1.1. Message Passing Interface Forum, University of Tennessee, Knoxville, June 1995. `http://www.mcs.anl.gov/Projects/mpi/mpich/index.html`.

[Nip89] Nipkow, T. Formal verification of data type refinement: Theory and practice. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop), volume 430 of *Lecture Notes in Computer Science*, pages 561–591, Mook, The Netherlands, May–June 1989. Springer-Verlag.

[ORR+96] Owre, S., Rajan, S., Rushby, J. M., Shankar, N., and Srivas, M. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Verlag, 1996.

[Pau93] Paulson, L. C. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[PPG+96] Petrov, T. P., Pogosyants, A., Garland, S. J., Luchangco, V., and Lynch, N. A. Computer-assisted verification of an algorithm for concurrent timestamps. In Gotzhein, R. and Bredereke, J., editors, *Formal Description Techniques IX: Theory, Applications, and Tools* (FORTE/PSTV'96: Joint Int. Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification), pages 29–44, Kaiserslautern, Germany, October 1996. Chapman & Hall.

[SAGG+93] Søgaard-Andersen, J. F., Garland, S. J., Guttag, J. V., Lynch, N. A., and Pogosyants, A. Computer-assisted simulation proofs. In Courcoubetis, C., editor, *Computer-Aided Verification* (5th Int. Conference, CAV'93), volume 697 of *Lecture Notes in Computer Science*, pages 305–319, Elounda, Greece, June–July 1993. Springer-Verlag.

[Smi97] Smith, M. *Formal Verification of TCP and T/TCP*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1997.

[Tau] Tauber, J. A. IOA code generation—theory and practice. Manuscript.

[vRBM96] van Renesse, R., Birman, K. P., and Maffeis, S. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.