The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems

Stephen J. Garland and Nancy A. Lynch
MIT Laboratory for Computer Science, Cambridge, MA

August 7, 1998
(original version: September 25, 1997)
Technical Report MIT/LCS/TR-762
MIT Laboratory for Computer Science

Abstract

This report describes a new language for distributed programming, the *IOA language*, together with a high-level design and preliminary implementation for a suite of tools, the *IOA toolset*, to support the production of high-quality distributed software. The language and tools are based on the I/O automaton model, which has been used to describe and verify distributed algorithms. The toolset supports a development process that begins with a high-level specification, refines that specification via successively more detailed designs, and ends by automatically generating distributed programs. The toolset encourages system decomposition, which helps make distributed programs understandable and easy to modify. It also provides a variety of validation methods (theorem proving, model checking, and simulation), which can be used to ensure that the generated programs are correct, subject to assumptions about externally-provided system services (e.g., communication services), and about the correctness of hand-coded data type implementations.

Keywords: Distributed systems, distributed algorithms, I/O automata, software tools, verification, theorem proving, simulation, code generation, model checking.

1 Introduction

I/O automata [58, 59] have been used to model and verify many distributed algorithms and distributed system designs, and also to express many impossibility results. See, for example, [50, 52, 40, 10, 13, 25, 26, 16, 71, 72]. The model has many features that make it suitable for such tasks: its fundamental concepts are mathematical (rather than linguistic); it is simple; it includes a notion of external behavior based on simple linear traces; it includes a notion of composition based on synchronized external actions; and it supports description of levels of abstraction, with successive levels related by inclusion of trace sets. The model supports a rich set of proof methods, including invariant assertion techniques, forward and backward simulation methods, and compositional methods. It provides excellent support for system decomposition. A concise exposition of the model and most of its proof methods appears in Chapter 8 of [50].

The model and methods were originally developed for reasoning about theoretical distributed algorithms, but, in the past few years, they have been applied increasingly to practical system services such as distributed shared memory services [25, 24, 23], group communication services [26, 27, 16, 19, 41], and standard communication services like TCP [72, 71, 73]. For these services, the theory has contributed system descriptions and proofs, and also important structural information such as interface specifications, invariants, and simulation relations. Ambiguities have been resolved, and several problems have been found, including logical errors in key algorithms in the Orca [8] and Ensemble [38, 39] systems and some unexpected behavior in the T/TCP communication protocol [11].

Most of this work has been done by hand; however, much of it (primarily, invariant and forward simulation proofs) is sufficiently stylized to admit computer assistance using interactive theorem provers such as the Larch Prover [28] or PVS [70]. For example, we have proved the correctness of a version of the Dolev-Shavit Bounded Concurrent Timestamp protocol [20, 30] using the Larch Prover [68]. Also, Archer has proved most of the invariants in our paper [27] on view-synchronous group communication, using the PVS theorem prover [5]. Of course, many other researchers have also used theorem provers to prove invariants and simulation relations for distributed systems, using other state-machine models and theorem provers as well as I/O automata and LP.

The success of this formal modelling/verification work for both distributed algorithms and distributed system services suggests that such techniques can play an important role in the development process for real distributed systems. However, there have so far been some serious barriers to their use. The main problem is that there is generally *no formal connection between verified designs and corresponding final code.* Although it is feasible to verify the correctness of an abstract distributed system design using a theorem prover, there is no convenient way to extend this proof to the final code. Rather, the verified design is generally re-coded in a real programming language like C++ or Java. This coding step involves costly duplication of effort and can introduce errors.

A related problem is the *lack of a programming language for distributed systems that is suitable for both verification and code generation.* It is not obvious what such a language should look like, because the features that make a language suitable for verification (axiomatic style, simplicity, nondeterminism) differ from those that make it suitable for code generation (operational style, expressive power, determinism). In previous work on I/O automata, descriptions were written in an informal pseudocode rather than in a real programming language. The pseudocode was hand-translated into the input language of a theorem prover, which was time-consuming and introduced error possibilities. Also, it was not clear how to convert this pseudocode to running code.

Another problem, for I/O automata in particular, is the *unavailability of light-weight validation tools* such as simulators and model checkers. Such tools can yield quick feedback to help in debugging, prior to attempting a time-consuming formal proof.

In this report, we propose a way of addressing these problems, in the form of a new language for distributed programming, the *IOA language* [29], together with a design for a coordinated suite of tools, the *IOA toolset.* The language and tools are based on the I/O automaton model and support proofs, light-weight validation, and code generation. The toolset is designed to produce efficient, running distributed programs whose correctness has been proved, subject to stated assumptions about the behavior of externally-provided system services (e.g., communication services), and subject to assumptions about the correctness of hand-coded data type implementations. The toolset also supports system decomposition, using shared-action parallel composition and levels of abstraction.

In the mode of operation we propose, the designer of a distributed system or algorithm would develop and validate his/her design entirely within the IOA framework. The final result—still an IOA program—would be translated automatically into source code in an existing programming language like C++, Java, or ML, thereby eliminating the need for a final coding step. Before the translation, the design would be subject to a range of validation methods, including complete proof using an interactive theorem prover, study of selected executions using a simulator, and exhaustive study of small instances of the design (and other debugging help) using a model checker. The

IOA toolset would assist the programmer in decomposing the design into separable interacting components, based on the formal I/O automaton composition operator, and in refining it using levels of abstraction, based on trace inclusion and forward and backward simulation relations.

This paper introduces the IOA language and toolset and discusses how they meet the design requirements for this combination of capabilities. The key components of our design are: the IOA language, the basic support for composition and levels of abstraction, the method of interfacing to existing theorem provers, the simulation method, the code generation method, and the method of interfacing to existing model checkers. We present requirements that we believe these various pieces must satisfy, our designs for these pieces, and some discussion of the advantages and disadvantages of our choices.

Interesting features of the language include its guarded command style, its use of nondeterminism, its handling of data types, its use of both axiomatic and operational descriptions, and its support for shared-action composition and levels of abstraction. Interesting features of the toolset include its support for levels of abstraction (e.g., user-specified step correspondences), certain aspects of the simulation method (e.g., restricted form for programs, paired simulations) and certain aspects of the code-generation method (e.g., restricted form for programs, abstract channels).

The IOA reference manual is available at
`URL http://larch-www.lcs.mit.edu:8001/~garland/ioaLanguage.ps`.
A parser and static semantic checker, which produce an internal representation suitable for use by validation and code generation tools, are available upon request to the authors. The rest of the tools have high-level designs, as described in this paper, and are currently in various stages of detailed design and implementation. In particular, Anna Chefter has produced a detailed design for the simulator [14], and a preliminary implementation, also available upon request.

There are many more possibilities for interesting work on this project than we can carry out ourselves, and we hope that others will become interested in pursuing some of them.

2 Related Work

It is hard for us to give adequate citations to related work, because there is so much of it. Many researchers and developers have worked on aspects of the general problem of modelling and validating distributed system designs, and the problem of generating code from higher-level formal descriptions. Many other toolsets for manipulating distributed programs exist. In this section, we will only mention what we think is the closest work.

The IOA language evolved from pseudo-languages used in research papers and books on distributed algorithms (see, e.g., [50, 52]). These pseudo-languages are based on named, parameterized *transition definitions* with *preconditions* and *effects* (that is, guarded commands). The effects code is either *operational* (an imperative program), or *assertional* (a predicate relating pre- and post-states). A similar precondition/effect style is used in other languages such as TLA [45] and UNITY [12], although these are based on somewhat different automaton models, as discussed below. Several proofs for distributed algorithms, modelled using I/O automata and an IOA pseudo-language, and using invariants and simulation relations, have been carried out using interactive theorem-provers. The first was Nipkow's proof for a reliable communication protocol [64] using Isabelle; other examples include connection management protocols, mutual exclusion algorithms, and bounded concurrent timestamp algorithms, using the Larch Prover [75, 48, 68], and group communication algorithms and simple hybrid systems using PVS [5, 6]. Some of these use "timed" extensions of the I/O automaton model [56] and pseudo-language.

Goldman's Spectrum system [31] includes a formally-defined programming language for describing I/O automata; it is based on transition definitions with preconditions and effects, but makes some different design choices from IOA. His language uses operational descriptions for transaction effects. He developed a (single machine) simulator for his language. The language was not connected to a theorem prover or a code generator, although a strategy for distributed simulation was suggested. This strategy involved expensive non-local synchronization for implementing some transitions. Goldman's more recent work on the Programmer's Playground [33] also includes a language with formal semantics in terms of I/O automata. Cheiner and Shvartsman [15] generated code for a specific I/O-automaton-based distributed algorithm—the Eventually Serializable Data Service of Luchangco et al. [24], and also gave some suggestions for more general distributed code-generation strategies. Their approach was not based on a formal programming language, and did not involve theorem proving. Their principal code-generation strategy requires non-local synchronization for some transitions, although they give some discussion of a local strategy.

Process algebraic languages such as CSP [42] and CCS [63] also have automaton models that are similar to our I/O automaton models, in particular, they use a notion of composition based on synchronizing external actions. (However, external behavior is not typically described just by traces, but includes extra information such as "refusal sets" to capture certain liveness issues.)Although the automaton models are similar, process algebraic languages have a very different style and syntax from our IOA language: they denote processes in concurrent systems by using algebraic expressions with rich sets of algebraic operators, starting from some basic single-step processes. Notation for describing *states* can be complicated. Some process algebraic languages admit rich sets of validation tools; for example, CCS is supported by the Concurrency Factory [17], which includes theorem provers, model checkers and simulators (but no distributed code generators). Proof methods tend to emphasize algebraic calculations, following the structure of the algebraic expressions denoting processes. The programming language Occam [1, 2] is based on the process-algebraic language CSP [42]. Occam code was compiled to run on Transputers, but it appears to us that no verification tools were provided for Occam programs.

Some other languages and tools for concurrent systems are based on different types of automata, which do not compose by synchronizing external actions, and do not use sets of traces as their notion of external behavior. For instance, Lamport's TLA language [45], Chandy and Misra's UNITY language [12], and Manna and Pnueli's language SPL [61] are based on automata that combine via shared variables instead of shared actions. (Actually, SPL automata also allow a special kind of CSP-style action synchronization.) We do not know of clear notions of external behavior for these models, analogous to sets of traces. In fact, work on these languages does not seem to emphasize formal parallel composition, which is basic to our work.

TLA, UNITY, and SPL are similar to IOA in that their basic program units are transition definitions having preconditions and effects. Effects in TLA are described axiomatically, while effects in UNITY and SPL are described operationally. These three languages support statement and proof of both safety and liveness properties, using temporal logic reasoning. The only tool we know of for TLA is the TLP theorem prover [21], based on Larch. A variety of UNITY validation tools have been developed, e.g., [4], and some UNITY code generation experiments have been carried out [77]. The Stanford STeP project [22] uses SPL. STeP has a powerful set of validation tools, including model checkers and theorem provers; we do not know of any STeP work on code generation.

Other related work includes that on Estelle [44] and other "Formal Definition Techniques" (FDTs). These are basically high-level, expressive programming languages with formal semantics, which can be used to describe or generate lower-level system code, and used as the basis for simulation. Proofs are not generally done for such programs, possibly because the expressiveness of the languages complicates the formal semantics. Also, we do not know of clear mathematical notions of external behavior for these models. Other related work includes that by Harel [37], Meseguer [62], and Ostroff [66].

3 The IOA Language

3.1 The I/O Automaton Model

The IOA language [29] is based on the *I/O automaton model* [58, 59]. I/O automata serve as models for reactive programs, which interact with their environments in an on-going manner. An I/O automaton consists of a set of *actions*, classified as *input*, *output*, or *internal*, a set of *states* (including a nonempty subset of *start states*), a set of *transitions*, which are (state, action, state) triples, and a set of *tasks*, which are sets of non-input actions. Input actions are enabled in all states. (Unlike in *synchronous* models such as SIGNAL [9], inputs do not trigger outputs.) The operation of an I/O automaton is described by its *executions*, which are alternating sequences of states and actions, and its externally-visible behavior is described by the *traces* of its executions, which are sequences of input and output actions. I/O automata admit a formal *parallel composition* operator, which allows an output action of one automaton to be performed atomically with input actions in any number of other automata; this operator respects the trace semantics. They also admit a *hiding* operator, which reclassifies output actions as internal so they cannot be used in further compositions. I/O automata admit a notion of *implementation* based on inclusion of sets of traces. Proofs for I/O automata typically involve *compositional reasoning*, *invariant assertions* (i.e., predicates that are true in all reachable states), and (forward and backward) *simulation relations*. An exposition of the model appears in Chapter 8 of [50]; simulation relations are presented in [55].

3.2 Requirements

First of all, we require that our language support precise and direct description of the mathematical model on which it is based. Since the I/O automaton model is a *reactive system model* rather than a sequential programming model, we expect the language to emphasize concurrency and interaction. In particular, we do not expect to start with a standard sequential programming language and simply add a few new constructs for expressing concurrency and interaction—these features should be at the core of the language. The language must also describe the notions of shared-action parallel composition and levels of abstraction precisely and directly.

Second, we require that the language be suitable for both verification (formal proof) and code generation. These two goals generate conflicting requirements: For verification, users[1] generally prefer a language that includes nondeterminism, so they can validate designs in as general a form as possible. Moreover, a simple language with an axiomatic style is easiest to translate into the input languages of standard theorem provers and easiest to manipulate in interactive proofs. In contrast, for code generation, programmers generally prefer a language with maximum expressive power. Moreover, a deterministic language with an operational style is easiest to translate into real code. We need a language that (somehow) satisfies both sets of requirements.

3.3 Design Decisions

The starting point for our design is the pseudocode used in the literature to describe I/O automata. This pseudocode contains explicit representations of automaton components (actions, states, transitions, etc.), where transitions are described using *transition definitions* (*TDs*) containing preconditions and effects. This pseudocode has evolved in two distinct forms: an *axiomatic style* [52], in which effects are described by means of predicates relating pre- and post-states, and an *operational style* [50], in which effects are described by means of programs using assignments, conditionals, and so on.

[1]In this paper, "user" denotes the user of the language and toolset, i.e., the programmer or program verifier.

To convert this pseudocode into a real programming language, we made several key design decisions:

1. We chose to define data types axiomatically, in the style used by the Larch Prover [28] and other theorem provers. This provides a sound semantics and facilitates translation into the input languages of theorem provers. We provide axiomatic definitions for built-in data types; the user may also define new data types, using Larch.

2. Since neither style alone is sufficient for all our purposes, we resolved the choice between axiomatic and operational styles for specifying effects of transitions by allowing both, either separately or in combination. That is, the effect for a TD may be described entirely by a program, entirely by a predicate, or by a combination, that is, by a program that includes explicit nondeterministic choices, followed by a predicate involving the post-state (and possibly also the pre-state) that constrains these choices. For example, we can write something like:

   ```
   x := choose a where 1 ≤ a ≤ 6
   y := choose a where 1 ≤ a ≤ 6
   so that x + y = 7
   ```

3. In the same spirit, variables are initialized using ordinary assignments and nondeterministic choice statements; in addition, the entire initial state may be constrained by a predicate.

4. Operational code in the effects of TDs has a very simple form, consisting of (possibly nondeterministic) assignments, conditionals, and simple bounded loops. This is reasonable because each transition is supposed to be executed atomically, making unbounded loops undesirable.

5. Automaton definitions and TDs can be parameterized, and the values of parameters can be constrained by predicates. TDs can have additional parameters, which do not appear in traces, but which allow values to be chosen to satisfy a precondition, and then used explicitly to describe its effects. (Spectrum [31] has some similar constructs.)

6. There is an explicit notation for composition, based on identifying external actions of different automata, and also an explicit notation for hiding output actions.

7. We introduced explicit notation to say that a predicate is an invariant of an automaton, and other notation to say that a binary relation is a forward (or backward) simulation from one automaton to another.

3.4 Discussion

The IOA language is very simple, allows precise and direct descriptions of I/O automata, and is designed to be used for both verification and code generation. It also allows formal description of system structure, using shared-action parallel composition and levels of abstraction. Using shared actions for composition allows us to define external behavior in terms of linear traces, which provide simple theoretical support for system decomposition. Evidence for IOA's expressiveness is that its pseudocode ancestors have proved adequate for modelling and verifying a wide range of algorithms and system designs (e.g., [50, 52, 40, 10, 13, 25, 26, 16, 71, 72]).

Nondeterminism is an important feature of the language, because it allows the programmer to avoid restricting his/her designs unnecessarily. Proving correctness of a design in its most general form is useful because the validity results can potentially be applied to many different implementations. Also, removing the "clutter" of unnecessary restrictions makes it easier to argue correctness, because it is easier to see what the correctness properties really depend on. (As in normal mathematical practice, we prefer to avoid cluttering up theorem statements with unnecessary hypotheses.)

IOA's guarded command style encourages programmers not to constrain the order of execution of actions unnecessarily, unlike standard sequentially-oriented languages, which we believe encourage overspecification of sequencing constraints. In highly interactive programs, especially at higher levels of abstraction, there are many situations in which the order of actions is not important for the properties of interest. (E.g., the order of processing elements from a set often does not matter.) Our philosophy of allowing maximum nondeterminism implies that such ordering constraints should be avoided. The nondeterministic guarded command versions of programs are more general, and are usually simpler and less cluttered, than their more sequential counterparts.

However, there are some situations in which some control over the order of actions is needed, particularly at lower levels of design, where performance considerations may force particular action scheduling decisions.[2] So far, IOA lacks explicit support (control structures) for describing the order of execution of transitions. In examples done so far, necessary ordering constraints have been expressed by introducing *pc* or *status* variables to keep track of progress in the sequential part of the computation. Although this method is adequate, it does not provide the same visual impact as more usual methods for describing sequencing, such as displaying lines of code to be executed in sequence vertically on a page, or displaying boxes with arrows in a flowchart.

So, at some point, it might be useful to add to IOA explicit support for specifying transition order. However, it is not yet clear how this should be done: standard control constructs used in sequential programming languages will not be sufficient, and some may not even be necessary. For example, reactive systems may contain threads that are intended to execute sequentially, but may be interrupted at any time; the interactions between the sequential threads and the interrupt-handling routines may be complicated to describe, and may require special control structures. On the other hand, the guarded command style allows alternative ways of describing iteration, which might mean that some standard looping constructs can be avoided. Flexibility may be important, to allow expression of many different kinds of control patterns.

If we do add such constructs to IOA, then to maintain simplicity and provability, and to ensure that we remain faithful to the underlying mathematical model, we will make sure that this addition is done as pure *syntactic sugar*, that is, so that there is an unambiguous translation of the code with the addition into code without it.[3]

The IOA language also has some minor limitations related to global vs. local naming. For instance, all of an automaton's state variables are global to all of its TDs; we may want to add variables whose scope is limited to a single TD. Also, all action names in a composition are global; we may also want to allow local action names, with a more flexible method of matching up names in a composition. At least, we will add a renaming operator for actions, which will yield most of the benefits of local action naming.

We plan to evaluate the desirability of modifying the syntax and semantics of the IOA language

[2]Sequencing of program steps within sequential programs implementing particular data type operations is a different issue, and is handled using standard sequential programming languages. This issue is discussed in Sections 5.3 and 5.4.

[3]For example, we can provide a way to represent a flow-chart with boxes corresponding to some of the TDs, with extra "places" representing *pc* values interspersed among the boxes; each box has exactly one input place, but the same place can be input to several boxes. Outgoing arrows from the boxes are labelled (exhaustively and exclusively) with state predicates. The translation to our basic language introduces an explicit *pc* value, an explicit statement at the end of the effect of each TD setting the *pc* as indicated by the predicate labels, and an explicit precondition for each TD saying that the *pc* indicates that place.

to reflect recent work on the Common Algebraic Specification Language (CASL) [65]. The current syntax and semantics of IOA is based on the Larch family of specification languages [36], which is one of the forerunners of CASL. CASL appears to provide better support than Larch for parameterized specifications. For example, CASL allows restrictions on data structures (e.g., that constrain the topology of a network) to be expressed in separate specifications, and those specification to be used as parameters to specifications for services that are guaranteed to work only when the restrictions are satisfied. CASL also has a richer type system than Larch, which makes it easier for designers to express abstractions.

Finally, an alternative approach to constructing verified code is to begin with a rich, expressive programming language, define a formal semantics and proof rules for that language, and attempt to use them for verification. The main problem with this approach is that complicated languages have complicated semantics and proof rules, which can be difficult for proof tools to manipulate, and (especially) difficult for users to think about. Artificial complexities introduced by such languages can become intertwined with the real complexities of the design being verified, making the job of verifying a complex design much more difficult than it needs to be. Instead, we have chosen to begin with a simple language that can express the concepts we need and that is amenable to formal proofs, and later add constructs (carefully) to the language to make life easier for programmers.

4 Example IOA Programs

In this section, we illustrate the design and use of IOA by developing a trivial banking system, in which a single bank account may be accessed from multiple locations. The account may be accessed using deposit and withdrawal operations (we assume that the balance can go negative), plus balance queries.

We give specifications for the system and its environment, in the form of two I/O automata `A` and `Env`. `Env` describes what operations can be invoked, where, and when; it may represent, for example, a collection of ATMs and customers interacting with those ATMs. Automaton `A` describes what the bank is alowed to do, without any details of the distributed implementation.

We also give a formal description `C` of a distributed implementation. We give a specification `B` for an intermediate service describing stronger guarantees on 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 constructs of the language.

4.1 Specification for a Banking Environment

The automaton `Env` specifies 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 each 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`, and in the case of a deposit or withdrawal, 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 at location `i`).

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] = 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
```

4.2 A Weak Specification of the Bank's Behavior

Our first specification, `A`, for the banking system describes what it can do, but not its implementation. Automaton `A` 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 contains 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 also parameterized by 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, the sequence number, and a `true` or `false` 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 Larch, 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. (Here, it is used to indicate that the return value has not yet been determined.)

The external signature of `A` is a "mirror image" of that of `Env`—its inputs match `Env`'s outputs and vice versa. `A` also has a single internal action `doBalance`, which calculates the balance for a balance query.

The state of `A` is described by 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 TDs are, for the most part, self-explanatory. The functions `insert` and `delete` are abstract functions defined (axiomatically) by the built-in data type `Set`. They are used to describe programs in which statements do not have side effects (thereby simplifying reasoning and complicating programming). The function `nat2pos` (defined in the auxiliary specification `NumericConversions`) converts natural numbers (elements of the built-in type `Nat`) to positive natural numbers (elements of 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 (at least) 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 the input language of the theorem prover we are using, the Larch Prover, is a first-order language without any special notations for set construction, the effect expresses a set inclusion using an explicit quantifier. Eventually, the IOA language may be enriched by more natural set notations, and the IOA toolset enriched to translate these notations into ones acceptable to particular theorem provers.

Finally, `reportBalance` reports any calculated, unreported balance to the environment.

```
automaton A(I: type)
  type OpRec = tuple of amount: Int, location: I, seqno: Pos, reported: Bool
  type BalRec = tuple of location: 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.location = 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.location = i ∧ y ∈ ops ⇒ y ∈ c) ∧ c ⊆ ops;
          bals := insert([i, define(totalAmount(chosenOps))],delete([i, null], bals))
```

The automaton AEnv is the parallel composition of the automata A and Env, matching external actions. This is written using the IOA language as:

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

The user can state invariants of the composition AEnv within IOA. In the following invariant, Clause 1 implies that the variable lastSeqno[i] is greater than or equal to all sequence numbers that have ever been assigned to operations originating at location i. Clause 2 implies that the sequence numbers assigned to operations submitted at location i form a prefix of the positive integers. Clauses 3 and 4 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. Clause 5 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.location])
  ∧ ∀ i:I ∀ k:Pos
       (   pos2nat(k) ≤ lastSeqno[i]
        ⇒ ∃ z:OpRec (z ∈ ops ∧ z.location = i ∧ z.seqno = k))
  ∧ ∀ x:OpRec
       (   x ∈ ops ∧ ¬x.reported
        ⇒    active[x.location]
           ∧ ∀ y:OpRec (y ∈ ops ∧ x.location = y.location / ¬y.reported ⇒ x = y)
           ∧ ∀ b:BalRec (b ∈ bals ⇒ x.location ≠ b.location))
  ∧ ∀ b:BalRec
       (   b ∈ bals
        ⇒    active[b.location]
           ∧ ∀ b1:BalRec (b1 ∈ bals ∧ b.location = b1.location ⇒ b = b1))
  ∧ ∀ x:OpRec ∀ y:OpRec
         (x ∈ ops ∧ y ∈ ops ∧ x.location = y.location ∧ x.seqno = y.seqno ⇒ x = y)
```

4.3 A Stronger Specification

Automaton B is very much like A, but imposes a stronger requirement 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]], by appending the statement:

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

to the transition definition for `requestBalance(i)`, and by modifying the `choose` statement in the transition definition for `doBalance(i)` to require the chosen set `c` of operations to include `mustInclude[i]`. In complete detail:

```
automaton B(I: type)
  type OpRec = tuple of amount: Int, location: I, seqno: Pos, reported: Bool
  type BalRec = tuple of location: 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],
    mustInclude: Array[I, Set[OpRec]] := constant({})
  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);
          mustInclude[i] := choose s where
             ∀ x: OpRec (x ∈ s ⇔ x ∈ ops ∧ x.reported)
    output OK(i)
      choose x: OpRec
      pre x ∈ ops ∧ x.location = 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.location = 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(type I); Env(type I)
```

It should be easy to see that the composition `BEnv` implements `AEnv` in the sense that every trace of `BEnv` is also a trace of `AEnv`. This can be shown formally using a trivial forward simulation relation from `BEnv` to `AEnv`—the identity relation for the state variables of `AEnv`. This relation can be expressed in IOA, as follows. This description uses a naming convention for variables in a composition that prefaces the name of each state variable in the composition with a sequence of names designating the automata of which it is a part. When there is no ambiguity, some of the automaton names in the sequence may be suppressed. For example, we can write `AEnv(type I).ops` or `A(type I).ops` in place of the complete name `AEnv(type I).A(type I).ops`.

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

4.4 Distributed Implementation

Now we describe a distributed implementation as an automaton `C` that is the composition of a node automaton `C(i)` for each `i` in `I`, plus reliable FIFO send/receive communication channels `channel(i,j)` for each pair of distinct indices, `i` and `j`, in `I`.

The channel automata are particularly simple: their state consists of a single variable, which which holds a sequence of messages. The keyword `const` in the signature indicates that the values of `i` and `j` in the actions of a channel automaton are fixed by the values of the automaton's parameters.

```
automaton channel(i, j: I, I, M: type)
  signature
    input send(m: M, const i, const j)
    output receive(m: M, 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)
```

Each node automaton `C(i)` keeps track of a set of operations that it knows about, including all 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 this automaton is particular to a location `i`, its action names are parameterized with `i`. Its `send` and `receive` actions are intended to match the corresponding channel actions. In the state of `C(i)`, `ops` is maintained as a set of records with no `reported` field; each record is an element of a new type `OpRec1`. The informaton about which operations have been completed is kept locally in a separate variable `reports`; it need not be 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 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 which is the union of the two individual types.

```
automaton C(i: I, I: type)
  type OpRec1 = tuple of amount: Int, location: 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.location = 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 C1 to be the composition of all the C(i) and all the channels, and C to be the same as C1 but with the communication actions hidden (this is to match the external signature of B).

```
automaton C1(I: type)
  compose C(i, type I) for i: I;
          channel(i, j, type I, type Msg) for i: I, j: I where i ≠ j

automaton C(I: type)
  hide send(m, i, j), receive(m, i, j) for m: Msg, i: I, j: I
  in C1
```

We define CEnv by composing C with the environment:

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

Now we have some invariants of CEnv: The first invariant says that any (deposit or withdrawal) operation that appears anywhere in the state (at a node or in a message) also appears in the ops set at its originating location.

```
invariant of CEnv:
     ∀ x:OpRec1 ∀ i:I (x ∈ C(i, type I).ops  ⇒ x ∈ C(x.location, type I).ops)
  ∧ ∀ x:OpRec1 ∀ i:I ∀ j:I ∀ m:Set[OpRec1]
       (   x ∈ m ∧ set(m) ∈ channel(i, j, type I, type Msg).queue
        ⇒ x ∈ C(x.location, type I).ops)
```

The next few invariants are analogous to invariants of AEnv.

```
invariant of CEnv:
     ∀ x:OpRec1
       (   x ∈ C(x.location, type I).ops ∧ ¬((x.seqno) ∈ C(x.location, type I).reports)
        ⇒    active[x.location]
           ∧ ∀ y:OpRec1
               (      y ∈ C(y.location, type I).ops
                        ∧  x.location = y.location
                        ∧  ¬((y.seqno) ∈ C(y.location, type I).reports)
                     ⇒ x = y)
             ∧  C(x.location, type I).bals = {})

   ∧ ∀ b:BalRec1 ∀ i:I
        (   b ∈ C(i, type I).bals
         ⇒ active[i]  ∧ ∀ b1:BalRec1 (b1 ∈ C(i, type I).bals ⇒ b = b1))
```

invariant of `CEnv`:

```
   ∀ x:OpRec1
      (x ∈ C(x.location, type I).ops ⇒ pos2nat(x.seqno) ≤ C(x.location, type I).lastSeqno)
```

invariant of `CEnv`:

```
   ∀ x:OpRec1 ∀ y:OpRec1
      (      x ∈ C(x.location, type I).ops  ∧  y ∈ C(y.location, type I).ops
          ∧  x.location = y.location  ∧  x.seqno = y.seqno
       ⇒ x = y)
```

invariant of `CEnv`: `∀ i:I ∀ k:Pos`

```
  (k ∈ C(i, type I).reports ⇒ pos2nat(k) ≤ C(i, type I).lastSeqno)
```

The next invariant is a trivial one, saying that channels from nodes to themselves are never used.

invariant of `CEnv`:

```
   ∀ i:I (   channel(i, i, type I, type Msg).queue = {}
          ∧  ¬C(i, type I).reqSent[i]
          ∧  ¬C(i, type I).reqRcvd[i]
          ∧  ¬C(i, type I).respRcvd[i])
```

Finally, we have an auxiliary invariant that is needed to prove some of the invariants of `CEnv1`. Claim 1 says that 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.) Claim 2 describes consistency among the sending and receiving flags. Claim 3 says that 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. Claim 4 says that if a response has been received, then a corresponding request was sent.

invariant of `CEnv`:

```
      ∀ i:I ∀ j:I ∀ s1:String
         (   req(s1) ∈ channel(i, j, type I, type Msg).queue
          ⇒     [null] ∈ C(i, type I).bals
             ∧  C(i, type I).reqSent[j]
             ∧  ¬C(j, type I).reqRcvd[i]
             ∧  (   channel(i, j, type I, type Msg).queue = {} ⊢ req(s1)
                 ∨  ∃ m:Set[OpRec1]
                         (   channel(i, j, type I, type Msg).queue = {} ⊢ req(s1) ⊢ set(m)
                          ∨  channel(i, j, type I, type Msg).queue = {} ⊢ set(m) ⊢ req(s1)))
             ∧  ¬∃ m:Set[OpRec1] (set(m) ∈ channel(j, i, type I, type Msg).queue)
             ∧  ¬C(i, type I).respRcvd[j])
```

```
∧ ∀ i:I ∀ j:I
     (   C(j, type I).reqRcvd[i]
      ⇒     [null] ∈ C(i, type I).bals
         ∧  C(i, type I).reqSent[j]
         ∧  ¬C(i, type I).respRcvd[j])

∧ ∀ i:I ∀ j:I ∀ m:Set[OpRec1]
     (   set(m) ∈ channel(j, i, type I, type Msg).queue
      ⇒     [null] ∈ C(i, type I).bals
         ∧  C(i, type I).reqSent[j]
         ∧  ¬C(j, type I).reqRcvd[i]
         ∧  (   channel(j, i, type I, type Msg).queue = {} ⊢ set(m)
             ∨  ∃ s1:String
                   (   channel(j, i, type I, type Msg).queue = {} ⊢ req(s1) ⊢ set(m)
                    ∨  channel(j, i, type I, type Msg).queue = {} ⊢ set(m) ⊢ req(s1)))
         ∧  ¬∃ s1:String (req(s1) ∈ channel(i, j, type I, type Msg).queue)
         ∧  ¬C(i, type I).respRcvd[j])

∧ ∀ i:I ∀ j:I (C(i, type I).respRcvd[j] ⇒ C(i, type I).reqSent[j])
```

To show that CEnv implements BEnv, we would like to define a forward simulation relation from CEnv to BEnv. In order to do this, it is convenient first to add "history variables" mustInclude[i], for i in I, to CEnv. The name mustInclude[i] is suggestive of the same-named variable in BEnv; there is a slight difference in that here mustInclude[i] is a subset of OpRec1 rather than of OpRec. These history variables are maintained by means of the following addition to the transition definition for requestBalance in CEnv:

```
requestBalance(i)
  eff
    ...
   mustInclude[i] := choose c where ∀ x: OpRec1
      (    x ∈ c
       ⇔ ∃ j:I
             (x ∈ C(j,type I).ops  ∧  x.location = j  ∧  x.seqno ∈ C(j,type I).reports) )
```

Let CEnv1 denote the result of augmenting the composition CEnv with these history variables. We can give some invariants involving the new history variables. In the first invariant, Claim 1 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 state. 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. Claim 2 is a variant of Claim 1 for operations originating at location i; it says that these operations must already be at location i. The second invariant just says that any operation in mustInclude[i] must have actually completed.

```
invariant of CEnv1:
     ∀ i: I ∀ j: I  ∀ x: OpRec1
        (      [null] ∈ C(i, type I).bals
           ∧  x ∈ CEnv1(type I).mustInclude[i]
           ∧  x.location = j
           ∧  j ≠ i
         ⇒     x ∈ C(j, type I).ops
```

```
            ∧ ∀ m: Set[OpRec1](set(m) ∈ channel(j, i, type I, type Msg).queue ⇒ x ∈ m)
            ∧ (C(i, type I).respRcvd[j] ⇒ x ∈ C(i, type I).ops))

  ∧ ∀ x: OpRec1
       (      [null] ∈ C(x.location, type I).bals
            ∧ x ∈ CEnv1(type I).mustInclude[x.location]
         ⇒ x ∈ C(x.location, type I).ops)
```

invariant of `CEnv1`:

```
  ∀ x:OpRec1 ∀ i:I
     (x ∈ CEnv1(type I).mustInclude[i] ⇒ x.seqno ∈ C(x.location, type I).reports)
```

Now we can define the desired forward simulation relation from `CEnv1` to `BEnv`. This uses a projection function `proj` from `OpRec`s to `OpRec1`s, defined in an auxiliary specification `Projections`, that just eliminates the `reported` component.

uses `Projection`

forward simulation from `CEnv1` **to** `BEnv`:

```
     BEnv(type I).active = CEnv1(type I).active
  ∧ ∀ x: OpRec
        (x ∈ B(type I).ops ⇔
              proj(x) ∈ C(x.location, type I).ops
            ∧ (x.reported ⇔ x.seqno ∈ C(x.location, type I).reports))
  ∧ ∀ x: BalRec
        (x ∈ B(type I).bals ⇔ [x.value] ∈ C(x.location, type I).bals)
  ∧ ∀ i: I (B(type I).lastSeqno[i] = C(i, type I).lastSeqno)
  ∧ ∀ i: I ∀ x: OpRec
        (x ∈ B(type I).mustInclude[i] ⇔
          proj(x) ∈ CEnv1(type I).mustInclude[i] ∧ x.reported)
```

5 The IOA Toolset

Now we describe the requirements for and the design of the IOA toolset. This is a high-level design; more detailed designs and implementations are in various stages of completion.

An overall design requirement is to have the tools based firmly on the underlying mathematical model. For instance, when we manipulate programs, we would like to be able to state theorems about the automata denoted by those programs. Another requirement is to keep the tools as simple as possible. We let the tools do the easy, routine jobs and rely on the user to help with the hard work. For instance, we rely on the user to help in many situations where a nondeterministic choice must be resolved.

We do not require that each of our tools be capable of processing the full IOA language; rather, we define special forms for programs (for the simulator, code generator, and model checker) and rely on the user to help in transforming programs into the special forms. The tools provide support in defining and verifying the correctness of these transformations.

Currently, the IOA tools deal only with safety properties. Instead of liveness properties, we plan to consider time bounds, which can be treated as safety properties. This will involve a future extension of our system to timed I/O automata [56, 60].

5.1 Basic Support Tools

5.1.1 Parser and static semantic checker

The IOA language has a working *parser* and *static semantic checker*, which produce an internal representation suitable for use by other tools. There is also a *prettyprinter*, which tidies up the code, e.g., by indenting it consistently, breaking long lines, and helping to format it using LaTeX.

Example 1 *The parser, static semantic checker and prettyprinter have been used to process all the code in the bank example in Section 4 except for the invariants and simulation relation involving* `CEnv1`*. Since the IOA tools currently provide no support for defining* `CEnv1` *by adding the history variable* `mustInclude[i]` *to* `CEnv`*, we checked the statement of these invariants and simulation relation against the automaton* `CEnv` *and received the expected error messages about* `mustInclude` *being undefined.*

5.1.2 Composer

A *composition tool (composer)* converts the description of a composite automaton into *primitive form* by explicitly representing its actions, states, transitions, and tasks. In the resulting description, the name of a state variable is prefixed with the names of the components from which it arises. Such prefixes may be abbreviated as long as no ambiguity is introduced. The input to the composer must be a *compatible* collection of automata; for example, the component automata should have no common output actions. This compatibility is checked using other tools—in simple cases, the static semantic checker, and in more complicated cases, the theorem prover, as indicated in Section 5.2.2.

Example 2 *In the bank example, the user can use the composer tool to produce intermediate IOA code for the compositions* `AEnv`*,* `BEnv`*, and* `CEnv`*.*

The design of the composer is described in more detail in [14]. A preliminary implementation has been developed and will be available soon.

5.1.3 Step correspondence

A novel part of our design is its support for programming using levels of abstraction. When users attempt to show that an automaton, B, implements another automaton, A, they generally expect to define a simulation relation—a predicate relating the states of B and A —for example as illustrated in Section 4.3. However, for some purposes, the predicate alone is not enough; the user must also supply information about the correspondence between *steps* of B and A. We think it reasonable for the user to do this, because the step correspondence expresses key facts about the relationship between the automata, which we think are useful for the user to understand.

The toolset helps users define step correspondences. Consider the case of a forward simulation R. (The handling of backward simulations is analogous.) Given a step (s_B, π, s'_B) of B (arising from a given TD), and a state s_A such that $(s_B, s_A) \in R$, the user must define an execution fragment of A that corresponds to the given step of B. One way in which the user might specify this execution fragment is by providing, as a function of the given step and state, (a) a sequence of TDs of A, plus (b) a way of resolving each explicit nondeterministic choice (i.e., those represented by **choose** statements and **choose** parameters) that appears in those TDs. (This information is enough to define the corresponding execution fragment of B uniquely.) The user will be able to describe this function by using a case statement to define the sequences of TDs and by supplying subroutines to resolve the nondeterministic choices.

It is not always obvious how the user can define the needed fragment of A solely as a function of the given step and state. For example, the choice of fragment of A (e.g., the resolution of the explicit nondeterministic choices in the fragment of A) may depend on explicit nondeterministic choices, or on which branch of a conditional is taken, in the step of B. In such cases, the user may want to modify the code of the given TD of B by adding history variables to record relevant choices, and then to use the values of these history variables in state s'_B in defining the execution fragment of A (e.g., in resolving the explicit nondeterminism in the execution fragment).

The IOA system will support modifying the code for B in this fashion, checking that the modifications are "allowable", that is, that they do nothing more than add new variables to the state and new statements that assign values to those variables. In particular, they may not change existing assignments. The toolset will also provide a little language for use in defining step correspondences.

These step correspondences can be used by our toolset in at least two ways: (a) In proving correctness of a simulation relation, using a theorem prover. (See Section 5.2.) (b) In testing whether a proposed simulation relation appears to be correct, using a simulator. (See Section 5.3.)

Example 3 *In the bank example, the simulation relation from* `BEnv` *to* `AEnv` *projects the state variables of* `BEnv` *onto the state variables of* `AEnv`*, and the step correspondence extends this projection to (state, action, state) triples. This provides all the information that is needed to verify the simulation relation using a theorem prover. However, it does not provide quite enough information to test the simulation relation using a simulator. In particular, the user needs to direct the simulator to make explicit choices in the steps of* `AEnv` *"in the same way" as in* `BEnv`*. This latter correspondence can be expressed using history variables.*

The interesting cases are `doBalance(i)`*, because of its use of a* `choose` *expression that makes a nondeterministic choice, and* `OK(i)`*, because of its use of a* `choose` *parameter. First, suppose the user is given the step* $(s_B, \text{doBalance}(i), s'_B)$ *of* `BEnv`*, and a state* s_A *of* `AEnv` *such that* s_B *and* s_A *are related. Then the step correspondence specifies that the sequence of actions of* `AEnv` *that will appear in the corresponding execution fragment is the single-action sequence* `doBalance(i)`*, and that the state* s'_A *that results from the execution of this action from state* s_A *is just the projection of* s'_B *on the state variables of* `AEnv`*. The user can now direct the simulator to resolve the nondeterministic choice* *in the* `choose` *statement in* `doBalance(i)` *in* `AEnv` *by using the same value that was chosen in the* `choose` *statement in* `BEnv`*. In this case, this value is recorded in the program variable* `chosenOps` *in state* s'_B*.*

Second, suppose the user is given the step $(s_B, \mathrm{OK}(i), s'_B)$ *of* `BEnv`*, and a state* s_A *of* `AEnv` *such that* s_B *and* s_A *are related. Again, the step correspondence specifies that the sequence of actions of* `AEnv` *is the single-action sequence* `OK(i)`*, and that* s'_A *is the projection of* s'_B*. Now the user needs to resolve the nondeterminism represented by the* `choose` *parameter* `x`*. The appropriate value for this choice is the value that was chosen for the corresponding* `choose` *parameter in* `BEnv`*. Unfortunately, this value, while a member of* `ops` *in states* s_B *and* s'_B*, is not readily identifiable in either state. However, the user can make this value available in state* s'_B *by adding an appropriate history variable to the automaton* `BEnv`*.*

Example 4 *The step correspondence for the simulation relation from* `CEnv1` *to* `BEnv` *is trivial for* `send` *and* `receive` *steps. For each other step of* `CEnv1`*, the fragment of* `BEnv` *consists of a single step with the same action. Since the forward simulation is a function defined in IOA, this step is uniquely determined, and its final state can be defined in IOA. Again, to direct a simulator, the user can describe how to resolve explicit nondeterminism in the same manner as in Example 3.*

5.2 Theorem Prover

5.2.1 Overview

A theorem prover can be used for IOA programs in several ways, for example, to prove validity properties for programs and other user inputs, to prove facts about the data types manipulated by programs, to prove invariants of automata, and to prove (forward and backward) simulation relations between automata.

The IOA toolset will contain interfaces to a number of existing theorem provers. The first such interface is targeted to the Larch Prover (LP) [28], which is based on first-order logic. The IOA manual [29] contains an overview of Larch. Devillers and Vaandrager have begun working on an interface to PVS [70]. Interfaces to HOL [34] and Isabelle [67] are also desirable. Although their input languages and proof capabilities differ, the differences between these theorem provers are unimportant for our purposes, because the types of things we need to express and to prove are within the capabilities of all standard theorem provers.

These interfaces will translate IOA descriptions of automata into axioms that can be used by the targeted theorem provers. These axioms will provide a mathematical description of the underlying automaton, defining its actions, states, transitions, and tasks explicitly. The IOA language has been designed to facilitate this translation. All the operational statements in TDs, including assignment statements, **choose** statements, conditionals, and loops, can be replaced by constraints expressed as predicates, allowing the transitions to be described as the set of (state, action, state) triples satisfying certain constraints. Similarly, the initial states can be described as the set of value assignments for all the variables satisfying certain constraints. All these constraints are axioms in the language of the theorem prover. Additional axioms are provided by the formal definitions of the data types used in the automata.

When a well-formedness condition for an automaton (e.g., that the set of choices for a nondeterministic assignment is always nonempty) is too hard to establish by static checking, the interfaces can formulate this condition as a theorem that must be proved. They can also formulate sets of lemmas that imply that asserted invariants and simulation relations are indeed invariants and simulation relations.

Users can interact directly with a theorem prover to prove that the lemmas follow as consequences of the axioms. In some cases, the theorem prover can prove a lemma automatically, using algebraic substitutions and other logical deductions. In other cases, the user must interact with the prover to suggest proof strategies and other useful information.

5.2.2 Validity of programs and other input

Some validity properties for IOA programs cannot be checked statically, but can be verified using the theorem prover. For example, IOA requires that a **choose** statement always choose from a nonempty set. Determining if this is so is undecidable in general, but in practice, it can often be proved using a theorem prover. The interface can translate the nonemptiness requirement into a lemma describing constraints on the set in a **choose** statement, again by converting operational statements into predicates. For another example, IOA requires that automata being composed be compatible, e.g., that they do not share any output actions. Since actions are described with parameters and **where** clauses, determining this is undecidable in general, but it may be proved using a theorem prover.

The theorem prover can also be used to validate other inputs provided by the user in the course of using the tools. For example, the user provides help to the simulator in the form of a function that is supposed to indicate the next action to simulate (see Section 5.3). The theorem prover can be used to verify that the proposed action is enabled in the current state.

5.2.3 Invariants

IOA allows a user to assert that a predicate P is an invariant of an automaton A. The interface can convert P into lemmas saying that P is true in all start states and that P is preserved by all of A's transitions. These lemmas are generally organized by cases, based on particular TDs. The theorem prover can attempt to prove these lemmas, with user guidance. The interface can also provide some guidance, for example, telling the theorem prover that proving an invariant involves considering separate cases for each TD. If the theorem prover succeeds in proving the theorem, the user knows that the invariant is true. If not, the theorem prover lets the user know where it is stuck, which allows the user to provide additional suggestions or correct errors.

Example 5 *We have used the LP theorem prover to prove the given invariants of* `AEnv`, `CEnv` *and* `CEnv1`, *by hand-translating the IOA code for these systems into LP's input language.*

In the proof of the invariants for `AEnv`, *we had to direct LP to skolemize the precondition for the* `OK` *action and then divide the proof into cases depending on whether the chosen* `x` *was the same or different from the* `x` *and* `y` *in the statement of the invariant. There were also other case splits for the* `OK` *action, as well as for other actions (e.g., case splits on whether or not certain operations recorded in the state were submitted at the location performing the action).*

The proofs for `CEnv` *were considerably longer than those for* `AEnv`.

5.2.4 Simulation relations

IOA allows a user to assert that a predicate R is a forward simulation (or a backward simulation) from automaton B to automaton A. The interface can convert the proposed simulation relation R, together with user-proposed invariants for B and A, into a set of lemmas expressing a relationship between the start states of B and A, a relationship between the steps of B and A, and (in the case of a backward simulation) a totality condition. Formal definitions appear, for example, in Section 1.4 of [29].

However, unlike in the case of an invariant, we are not ready at this point to hand the problem off to the theorem prover. This is because some of the proof obligations for simulation relations contain existential quantifiers that the user must help to instantiate. For example, the "step" proof obligation for a forward simulation says that, for each step of B and each state of A corresponding to the pre-state, *there exists* a corresponding execution fragment of A. Proving such a statement automatically would be prohibitively difficult for a theorem prover, because it involves an expensive search. So, we allow the user to provide an explicit mapping from steps of B and states of A to execution fragments of A, as described in Section 5.1. The user also supplies help for other occurrences of existential quantifiers; for example, for a forward simulation, he/she specifies the corresponding start state of A for a given start state of B.

Now the theorem prover can attempt to prove the lemmas, with guidance from the user and interface. For example, for the step correspondence, the theorem prover can try to verify that the specified sequence is indeed an execution fragment, that it has the same external behavior as the given step, and that the final states are related by the proposed relation. Our experience with such proofs suggests to us that the step correspondence provided by the user is just what is needed to allow the theorem prover to carry out this work with a minimum of user interaction. If the theorem prover succeeds, the user knows that the simulation relation is correct, and if not, the user gets feedback to help complete the process.

Example 6 *We have used the LP theorem prover to show that the claimed relation is a forward simulation from* `BEnv` *to* `AEnv`*, again by hand-translating the IOA code into LP's input language. Informally speaking, it is easy to see that initial states are related, and that steps correspond as needed. The only significant difference is that the choice of a set of operations (in* `doBalance`*) is more restrictive in* `BEnv` *than in* `AEnv`*; the definition of a forward simulation allows for such restrictions.*

In order to guide the theorem prover in proving this theorem, we specified, for each step of `BEnv`*, the corresponding execution fragment of* `AEnv`*. We did this manually by means of a step correspondence, as described in Example 3. In particular, we provided the step correspondence for* `doBalance` *and* `OK` *steps by defining s'_A to be the projection of s'_B onto the state variables of* `AEnv`*. The proof of the simulation relation required no user guidance other than to provide the step correspondence.*

Example 7 *We have also used the LP theorem prover to show that the claimed relation is a forward simulation from* `CEnv1` *to* `BEnv`*. The interesting case in this proof is the* `doBalance(i)` *action. When this action is performed in* `BEnv`*, a set of operations is selected, including all those in* `BEnv.mustInclude[i]`*, which represents the deposit and withdrawal operations completed before the balance is requested. The first invariant of* `CEnv1` *implies that the choice that is made for this action in* `CEnv1` *includes* `CEnv1.mustInclude[i]`*, which implies that the relation is preserved.*

This theorem implies that `CEnv1` *implements* `BEnv`*. But it is also easy to show that* `CEnv` *implements* `CEnv1`*, so it follows by transitivity that* `CEnv` *implements* `BEnv`*, as we wanted to show.*

5.2.5 Discussion

Our experience indicates that the IOA language is easy to connect to a theorem prover. The main task, which is to convert programs from a mixed operational/axiomatic style to purely axiomatic style, is eased by the simplicity of IOA programs. If IOA is later extended with constructs to describe sequencing, this will be done as syntactic sugar, which will preserve the ease of translation into theorem-prover languages.

Although some proofs still require more user interaction than we would like, we believe that we can shorten such proofs (without waiting for better theorem provers to appear) by continuing to identify key forms of user input—primarily, resolution of nondeterministic choices—and by supporting the user in providing such inputs. For example, our experience confirms our belief that a user-provided step correspondence goes a long way toward allowing theorem provers to establish simulation relations with a minimum of user interaction.

Although carrying out full proofs for distributed algorithms using a theorem prover is now quite feasible, it is still painstaking and time-consuming work. Consequently, there is sill a need for lightweight validation tools such as a simulator or model-checker.

5.3 Simulator

5.3.1 Overview

A simulator [14] is required to run selected executions of an IOA program on a single machine. The user should be able to help select the executions that are run. The simulator should be able to check proposed invariants in the selected executions, provide some information about time performance, and check proposed simulation relations.

5.3.2 Special form for programs

We do not require that the simulator be capable of simulating arbitrary IOA programs, but expect users to transform programs first into a restricted form. The biggest problem faced by a simulator of distributed algorithms is resolving nondeterminism. IOA programs allow two kinds of nondeterminism: *explicit nondeterminism*, which arises from **choose** statements, **choose** parameters, and **choose** expressions in initial assignments, and *implicit nondeterminism*, which involves the scheduling of actions. The special form does not allow either type of nondeterminism.

Specifically, we assume that an IOA program submitted to the simulator satisfy the following restrictions:

1. It is in primitive form (a single automaton, with explicit description of its actions, states, transitions and tasks).

2. It is *closed*, that is, it has no input actions.

3. It has no explicit (`choose`) nondeterminism; thus, effects and initializations are defined operationally rather than axiomatically.

4. (To handle implicit nondeterminism) At most one non-input action is enabled in any state. Moreover, the user must specify what that action is, as a function of the state. (The output of this function is a TD plus expressions for the values of the parameters.)

5. All relations and functions are computable (e.g., all quantifiers range over finite sets).

The toolset will provide support to the user in getting programs into the required form. For example, the existing composition tool helps close an automaton, once the user provides a description of an "environment automaton" that supplies inputs for the given automaton, and helps transform representations using **compose** into primitive form. To resolve explicit nondeterminism, the system can prod the user to provide explicit choices to replace the nondeterministic choices (e.g., by adding a state variable representing a pseudo-random sequence and by replacing the nondeterministic choice by the value of a function applied to the next element of this sequence). If necessary, the theorem prover can be used to check that the provided choices satisfy any required constraints (expressed by **where** clauses, preconditions, or **sothat** predicates). Explicit nondeterminism can also be resolved by user input during the simulation or by system-generated probabilistic choices, but these mechanisms are merely syntactic sugar for transformations of a program into special form.

To remove implicit nondeterminism, the system can prod the user to constrain the execution of an automaton, e.g., by adding state variables containing scheduling information, adding extra constraints involving the new scheduling variables to the preconditions of actions, and adding new statements to the effects of actions to maintain the scheduling variables. The system can also ask the user to provide a function indicating the next action to be simulated; the theorem prover can be used to verify that the indicated action is in fact enabled, and that no other actions are enabled.

Such modifications may remove implicit nondeterminism entirely, or convert it into explicit nondeterminism, which can be resolved as described above. The following examples illustrate complete removal of implicit nondeterminism.

Example 8 Round-robin scheduling: *If a set of actions is to be executed in round-robin fashion, then a scheduling variable might keep track of the (index of the) next action to be performed, the precondition of each action might be augmented with a clause saying that the indicated action is the one recorded in the new variable, and the effect of each action might increment the index maintained in the new variable. This removes all the scheduling nondeterminism, and an obvious function of the state describes the next action to be performed.*

Example 9 Random scheduler: *A scheduler that uses randomness can also be described in the same general way. The system can allow the user to describe (as a function of the current state) a collection of possible next actions with associated probabilities, and can augment the automaton's state with a pseudorandom sequence, add a new constraint to the precondition of each action saying that it is selected by the next pseudorandom number, and add new statements to the effects that discard used pseudorandom numbers. This again removes all scheduling nondeterminism. The system could further assist the user in coming up with the probabilities for each action, by allowing the user to provide estimates of the* time *for each action. The system could then determine probabilities for the actions based on the user time estimates (with actions with low time estimates having correspondingly high probabilities).*

It is also possible to postpone resolution of some of the explicit or implicit nondeterminism until simulation time. For example, in the probabilistic scheme described above, calls to the pseudorandom number generator could be made on-line. Or, explicit nondeterminism could be resolved by user input or system-generated probabilistic choice. However, in these cases, the description given above is still a good conceptual view.

5.3.3 Main simulator loop

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 programming language (currently, Java). At present, we do not address the problem of establishing the correctness of this sequential code (other than by conventional testing and code inspection). For operations (such as `totalAmount`) defined in auxiliary Larch specifications, the user can choose to write the auxiliary specifications in an executable algebraic style, or to supply separate code libraries that purport to implement the specified operations. Standard techniques of sequential program verification based, for example, on Hoare logic, should be capable of handling such correctness proofs. (Note that the IOA framework focuses on correctness of the concurrent, interactive aspects of programs rather than of the sequential aspects.)

With all these assumptions, the simulator has a relatively easy job. Since all implicit nondeterminism has been removed, the simulator can start from the unique initial state and perform a loop in which, at each iteration, it executes the unique action enabled in the current state. More specifically, 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 checks that proposed invariants are indeed true in all states that arise in the simulated executions.

Example 10 *A restricted version of* `AEnv` *can be simulated. In this restricted version, the environment is constrained to submit particular operations in a particular way, e.g., round robin by location, with some particular pattern of deposits, withdrawals, and balance queries. The bank is constrained to calculate balances and respond at certain times, e.g., to process each request immediately, before the next request arrives. Explicit nondeterminism in the* `doBalance` *action can be resolved, e.g., by saying that the action returns all the operations up to the end of the last round-robin round. Also,* `I` *must be made explicit.*

By varying these restrictions, the simulator can simulate a wide range of schedules. The simulator can check all the invariants given for `AEnv` *in all states of the simulated executions.*

The simulator can also provide some performance information, by calculating "times" at which events of interest occur. It can do this by using upper bounds, provided by the user, for the time for steps in each task of the automaton; an on-line calculation can determine the greatest time at which each event can occur, subject to the task bounds.

5.3.4 Paired simulations

The simulator can also be used to check that a candidate relation R appears to be a simulation relation from B to A.[4] Namely, it can simulate B in the usual way and use the steps that arise in the simulation, together with a user-specified step correspondence, to generate a simulated execution of A. The simulator can check that the relation R holds after each step of B. We call this strategy a *paired simulation*.

For example, suppose that R is a proposed forward simulation relation. After the user specifies the mapping from steps to execution fragments (plus a mapping from start states of B to start states of A), the simulator can run the two algorithms together, letting the execution of B drive that of A. Specifically, after the simulator determines an initial state of B, it applies the user-provided mapping to get the initial state of A. Then the simulator simulates steps of B; for each such step, the simulator first simulates the step as usual, then determines the corresponding (proposed) execution fragment of A and runs the steps of that execution fragment. While it runs those steps, the simulator performs various tests, including: checking that the preconditions of all the actions are satisfied (with the user-provided values replacing the **choose** parameters); checking that the user-provided values used in **choose** statements and initial assignments satisfy their **where** clauses and **sothat** constraints; checking that the execution fragment has the same external behavior as the given step; and checking that relation R holds between the final states of the two automata after the step and the execution fragment. All of these can be done by evaluating predicates. Note that, although B must be described in the restricted input language, A need not be.

[4]We have an unfortunate clash of terminology here, between "simulator" and "simulation relation".

Paired simulations can also help in checking proposed backward simulation relations. However, in this case, the simulator first simulates a (finite) execution of B completely, then begins at the end and generates the corresponding execution of A backwards.

5.3.5 Discussion

The simulator has an easy job because all nondeterminism is removed ahead of time. The system provides support in removing nondeterminism, in the form of general support for transforming automata (e.g., using levels of abstraction), and special support designed for use with the simulator (e.g., suggestions to remove **choose** constructs, and help in building randomized scheduling policies into the state of an automaton).

The use of the user-provided function should yield more efficient schedules than would arise from a general scheduling discipline, and should also help to avoid some runtime overhead.

We expect that the paired simulation strategy will prove to be a very useful tool for system development using levels of abstraction.

A detailed design for the simulator appears in [14], and a preliminary implementation has been carried out.

5.4 Code Generator

5.4.1 Overview

The code generator is required to generate real code for a distributed system. The target system should be flexible—an arbitrary configuration of computing nodes and communication channels. The generated code should be efficient enough to use in real applications.

The process of generating code remains, as long as possible, within the IOA framework. Design proceeds as usual, possibly in several levels of abstraction, until the final IOA description conforms closely to the available hardware and software services. This final IOA description consists of a collection of *node automata*, one for each actual machine, and *channel automata* for externally-provided communication services. Then each node automaton can be translated automatically into actual code that implements the node automaton, in the sense of trace inclusion. Each channel automaton can be implemented, also in the sense of trace inclusion, by an externally-provided service such as TCP [69] or MPI [35, 3]. (However this implementation guarantee might only hold with high probability, e.g., it might be conditional upon certain assumptions about the behavior of the underlying hardware.)

This process allows the IOA formal modelling and verification facilities to be used to reason about the design until the last possible moment. It can be used to ensure that the final system provably implements higher-level IOA descriptions, subject to assumed properties of the externally-provided services, the handwritten data type implementation code, and the underlying hardware.

5.4.2 Special form for programs

In addition to restricting IOA programs to have a special form that matches the distributed system architecture, the code generator requires that node programs satisfy restrictions similar to those required by the simulator. Specifically, programs should be in primitive form, and should be devoid of both explicit and implicit nondeterminism, but they need not be closed. (Thus, a node program is essentially a sequential program consisting of non-input actions, with the possibility of interrupts from input actions.) As before, the user must 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 part of each transition be done without interruption, even if inputs arrive from the external user or from the communication service during its execution. In our design, such inputs are buffered. In between running non-input actions, the generated program examines buffers for newly arrived inputs, and handles (some of) them by running code for input actions. (At each examination point, it might run all inputs, or just one, or one from each input "port", etc.) Since the processing of inputs is delayed (with respect to the performance of these actions by their originators), such delays have the potential to upset precise implementation claims for the node automata.

Example 11 *Consider a node automaton that receives inputs a and b, in that order, from two different parts of its environment, If the implementation does not perform the inputs immediately, but instead buffers them in separate buffers, polls the buffers, and processes the inputs after some delay, it might process the inputs in the reverse order b, a. An execution thereby occurs in which the processing order is different from the order in which the events are received from the environment, which violates the correctness of the implementation.*

We introduce a restriction on the node programs that prevents such delays from upsetting the implementation claims. Stated in its strongest, simplest form, the 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. Formally, $traces(A' \times Buff)$ must be a subset of $traces(A)$, where A' is just 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". It is possible to weaken this requirement slightly, to require trace inclusion only in "well-formed" environments (e.g., only for blocking inputs); then these assumptions must be satisfied by the actual environments of the node automata. Or, we can allow a "port" structure for different types of inputs, corresponding to actual arrival ports for inputs, and assume that *Buff* preserves the order of inputs on each port.

As for the simulator, the system can provide some help in getting the program into the special form. The general facilities to support programming using levels of abstraction can be used to refine the design within the IOA framework until the required node-and-channel form is reached. Support for putting the program into primitive form and for removing explicit and implicit nondeterminism is the same as for the simulator. IOA specifications for actual communication services like TCP and MPI are maintained in a library; we obtain such specifications by reading manuals and experimenting with available implementations. We do not generally try to prove that these specifications are correct descriptions of the real services; however, our methodology could be used to prove such results, based on modelling the protocols and hardware that implement the services. For input-delay-insensitivity, the user can verify the needed trace inclusion property using the tools that support programming in levels of abstraction.

5.4.3 Main code generator loop

The code generator uses the same library of data type implementations as the simulator. Again, we do not address the problem of proving the correctness of this sequential code. For each node automaton, the code generator translates the IOA code into a program in a standard programming language (like C++, Java, or ML) that performs a simple loop, similar to the one performed by the simulator. (In each iteration, the program uses the user-provided function to determine the next non-input TD and parameter values, then performs that TD with those parameter values.)

Note that it is possible for the code generator to translate the code at different nodes to different languages.

By insisting that IOA programs from which we generate real code match the available hardware and services, and by requiring the node programs to tolerate input delays, we avoid the need for expensive non-local synchronization in achieving a faithful implementation.

5.4.4 Abstract channels

Prior to using the code generator, it is often helpful to describe a system design as a composition of node automata A_i and *abstract channel automata* $C_{i,j}$, as illustrated in Figure 1. Such abstract channel automata are typically simpler than the automata for the real channels; for example, the `channel` automata in Section 4.4 might be used.

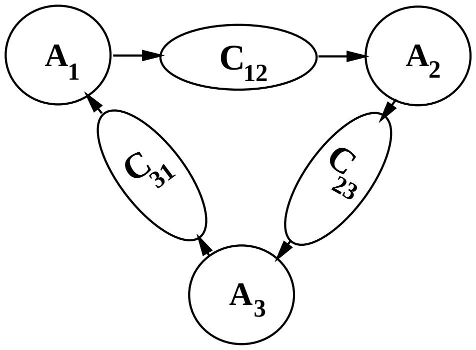

Figure 1: An implementation using abstract channels.

At a lower level of design, illustrated in Figure 2, each abstract channel automaton $C_{i,j}$ is implemented in terms of automata $D_{i,j}$ and $D_{j,i}$ representing real channels, together with protocol automata attached to the node automata. At this level of design, a node automaton is (formally) the composition of the node automaton from the previous level (the automaton that interacts with the abstract channels) with all the protocol automata for the node that appear in the channel implementations. It is this composed automaton (transformed into primitive form, and with its nondeterminism removed) that gets translated by the code generator into a standard programming language. The composed automata are circled in the figure.

Example 12 Abstract channel implementation *A send/receive channel from i to j can be implemented in terms of a single MPI service connecting i to j [78]. The implementation at the sender buffers the client's messages, sending them one at a time using MPI. At the receiver, the implementation does repeated non-blocking "probes" until it discovers that some message has arrived, then does a "receive" to retrieve the message and ships it to the client.*

The abstract channel strategy provides flexibility in that it allows different abstract channels to be used to describe implementations using the same distributed system architecture, and allows the same abstract channels to be used to describe implementations using different real channels.

The IOA system supports programming with abstract channels by maintaining libraries of IOA descriptions of abstract and real channels, and of IOA implementations of abstract channels in terms of real channels. The system also assists in proving correctness of implementations of abstract channels.

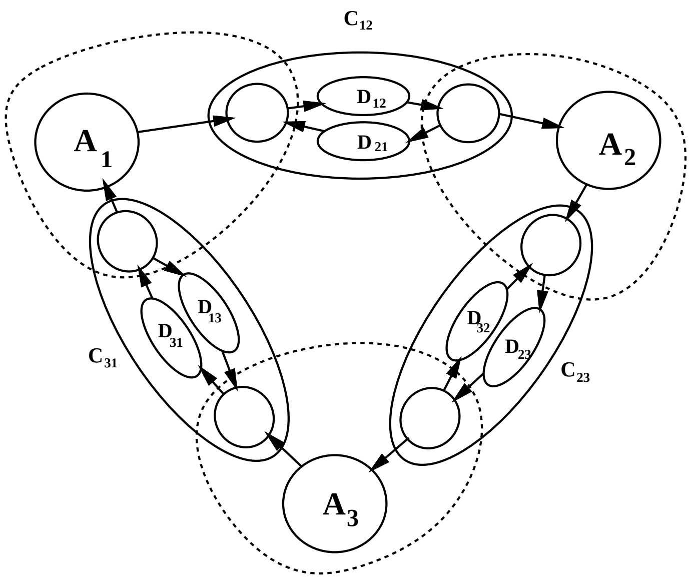

Figure 2: An implementation using abstract channels implemented by real channels.

Example 13 *The IOA system can be used to generate a running implementation of automaton* `C` *(that is, all except the environment) in the* `CEnv` *system, based on MPI. This strategy uses abstract send/receive channels, like those in Section 4.4.*

5.4.5 Discussion

This code generation method follows our general strategy of relying on the user (with support from IOA tools) to put the program in the proper form for input to the code generator. Once the program is in that form, the job of the code generator is not hard. We achieve provability by working entirely within the mathematically-based IOA framework until the last possible moment, then translating into real code using a very simple strategy.

Some of the real system services that we may want our IOA programs to interact with may be complicated and difficult to model accurately. We do not consider this to be mainly an IOA issue, but a general problem for system development. Our contribution is to provide a simple structure for modelling and reasoning about such services. In particular, we think that our simple trace-based notion of external behavior provides a good basis for modelling interfaces.

It must still be demonstrated that code with realistically good performance can be obtained using IOA. Some of our reasons for believing this will work are:

1. Our strategy achieves efficiency by working locally, without the need for any synchronization involving more than one machine. In contrast, the design sketched in [31, 32] and the main design in [15] use global synchronization strategies.

2. The code-generation process incorporates existing services (e.g., communication services), which may be highly optimized.

3. We allow hand-coding of data type implementations in a standard sequential programming language, rather than trying to generate these automatically, as is done in work on executable

specifications (see, e.g., [62]). This strategy provides many opportunities for hand optimization. If we like, these operations can be quite high level and powerful (e.g., a complicated function like a Fast Fourier Transform), as long as they can be computed by sequential code in the target programming language. These implementations can use powerful and complex control constructs, even recursion, as long as we are happy with the performance that results by performing the operation atomically at runtime.

4. Giving the user flexibility in controlling the order in which actions are performed should yield more efficient schedules for local node programs than would arise from a general scheduling discipline. Also, allowing the scheduler to call a user-provided function to determine the next action should avoid some runtime overhead.

   We believe that user control over the scheduler is even more important for the code generator than for the simulator. We believe that an automatic, general scheduling discipline can lead to excessive runtime overhead. Besides, we cannot now think of any general scheduling discipline for I/O automata that will always yield efficient choices of actions.

5. Source-to-source translation to C++ and other languages should be able to take advantage of prior work on optimizing compilers for those languages.

6. We should be able to avoid the expense of checks that would typically be done at runtime, by proving some properties statically.

7. IOA is sufficiently flexible to be used at different levels of abstraction, including a very low level that can permit extensive optimization within IOA itself.

8. Some work we have recently carried out with Hickey and van Renesse at Cornell [41], on modelling and validation of the Ensemble system using IOA-based methods, shows that it not hard to write and verify algorithms in IOA in such a way that they can be translated into ML code that is "essentially isomorphic" to the actual running code for the Ensemble system [39]. The existing ML code appears to run acceptably fast.

At later stages of development, we might want to try to improve performance by taking advantage of multithreading possibilities in the target programming language. This will involving loosening the restrictions on the programs that are input to the code generator, for example, allowing several actions to be enabled at a time. Maintaining action atomicity then becomes more complicated; concurrency control issues will need to be addressed here.

Also at later stages, we might want to provide some special default schedulers, which can allow the user to avoid specifying the next action. Because coordination and communication aspects of distributed computation are costly, it is possible that the extra overhead of such a scheduler might not be critical. We also would like to provide the user with convenient patterns for describing efficient schedulers, perhaps utilizing the "task" structure of the I/O automaton model [58, 59, 50], and perhaps with restrictions on the input programs.

Another programming issue that needs attention is the memory model—so far, the conceptual model involves infinite state machines with statically-allocated infinitely memory, but reality imposed some restrictions. Also, in our design, we have mostly ignored fault-tolerance issues. Good ways of modelling faults and faulty behavior within IOA need to be devised and incorporated into the toolset design.

5.5 Model Checker

5.5.1 Overview

Model checkers provide an approach to validation that is complementary to theorem proving and to simulation. Unlike theorem provers, model checkers work completely automatically. Simulators check a particular execution of a system, whereas theorem provers are used to validate all executions. Model checkers can be used to validate all executions of a finite system, provided that it has a sufficiently small number of states. They are often used to validate abstract finite versions of more complex infinite-state systems.

At present, the IOA system uses a model checker only to check invariants. The IOA toolset is designed to utilize existing model checkers. The first model checking interface is targeted to SPIN [43], with input language PROMELA. An automaton to be tested is represented as a single PROMELA process.

5.5.2 Special form for programs

We assume that an IOA program provided as input to the model checker is closed and in primitive form. It may be nondeterministic, but cannot have any **sothat** statements. Data types are restricted to those in the model checker's input language; for example, PROMELA allows integers, Booleans, and arrays of these. The program must have a fixed finite number of variables (although these variables may potentially take on infinitely many values).

The user provides an IOA program in the special form, and expresses the invariant as a predicate using IOA notation. The interface translates the program and predicate into the model checker input language. Then the model checker performs its work (with no user guidance). If it succeeds in verifying that the predicate holds in all reachable states, the user can infer that the predicate is an invariant of the original IOA program. If the model checker finds a counterexample, the user knows that the predicate is not an invariant and can use the counterexample for debugging. In our initial design, this counterexample is given in the language of the model checker, but later, we expect to add more system support so that the error is expressed in terms of the original IOA program.

Example 14 *In the bank example, the user can use the model checker to check the invariants in restricted cases of the system.*

5.5.3 Levels of abstraction

We would like to extend our notion of paired simulations to a model-checking technique for simulation relations; this remains to be done.

6 Extensions

We plan to extend the language and toolset to timing-based systems, using timed I/O automata as described in [56, 60] and Chapter 23 of [50]. This model has an external behavior notion based on timed traces, and supports parallel composition and hiding operators. It also admits reasoning using invariant assertions and various kinds of simulation relations.

This extension will support modelling and proof of timing-dependent distributed systems, and timing analysis for timing-dependent and non-timing-dependent distributed systems. The main addition to the model is that timed automata include time-passage actions as well as ordinary discrete actions. Safety properties of the usual kinds can still be stated and validated. Among the properties to be validated are those involving proofs of time bounds. Some of these properties will be conditional, based on assumptions about inputs to the algorithm and about failure patterns. (For this purpose, failures must be modelled explicitly, in terms of special "failure actions".)

It should be easy to extend the IOA language to a new *TIOA* language that includes time-passage transitions, described by TDs with preconditions and effects. For example, $\nu(t)$ may indicate time-passage by real time t. These actions are not classified as input, output or internal, but rather form a fourth category of actions. Among the data types to be used with this language are a "real number" data type, satisfying appropriate axioms (e.g., [47]), which can be used to describe quantities representing times. Time-valued variables can be included in an automaton's state. Typical such time-valued variables are a *now* variable represeng the current time, *last* variables representing deadlines for scheduled events, and *first* variables representing earliest times that certain events may occur.

Time-passage transitions are defined by transition definitions similar to the ones used in basic IOA. These include preconditions and effects. Now the preconditions and effects may involve the time-valued variables, as well as other variables. Typical preconditions involve things like: $\mathit{now} + t \leq \mathit{last}(e)$, which indicates that time cannot pass beyond the deadline for event e. Typical effects involve incrementing *now* by t. In addition, other actions may have preconditions involving time, for example, $\mathit{first}(e) \leq \mathit{now}$, and may modify the deadline variables.

TIOA admits explicit statement of invariant assertions and simulation relations. Both of these kinds of statements can involve time-valued variables; this allows them to be used to express time bounds and other timing relationships. Typical simulation relations for timed systems may involve inequalities between time-valued quantities in the specification and implementation automata. Proofs for such statements follow the same methods as for the untimed versions. Deduction involving inequalities plays a prominent role. Chapter 23 of [50] contains relevant definitions, and examples of usage of some pseudocode on which TIOA is based. Other examples appear, for instance, in [47, 76, 7, 48, 74, 46].

We expect that much of the design of the toolset will also carry over to TIOA. For example, a theorem prover can be used to prove invariant assertions and simulation relations for TIOA, using formalized versions of techniques used in [47, 76, 7, 48, 74, 46]. A prover should be able to handle the full TIOA language.

In order to prove conditional time bounds for a system expressed as a TIOA program, the user can formulate the restrictions involved in the conditional as a restricted version of the program, itself expressed as a new TIOA program. (For example, to consider the behavior of the program when there are no failures, one would restrict the program to omit explicit failure actions. Or, to consider the behavior after low-level timing behavior stabilizes, one would model the stabilization point explicitly, and restrict the program to behave in a restricted fashion after the stabilization point. Some ideas for how to do this appear in [18].) Then the conditional bounds can be proved for the restricted program.

A code generator for distributed code can be constructed, using basically the same strategy that we are using for the untimed case. Now the models for the channels are TIOA programs (timed automata), which include timing assumptions about the behavior of the channels; we rely on the externally-provided channel implementations to guarantee those assumptions. The node programs are also TIOA programs. The scheduler for a node program works essentially as in the untimed case, with some extra care to process inputs in a timely manner. For instance, after each selection and performance of a locally-controlled action, the scheduler may allow all pending inputs to be processed, before going on to the next locally-controlled action.

The user of the TIOA tools should prove that a proposed distributed algorithm, expressed in terms of node and channel TIOA programs, meets its timing requirements. However, now an additional step may be needed to demonstrate that the actual node implementation meets the timing requirements expressed by the higher-level node TIOA programs. Such a demonstration may depend upon modelling the low-level implementation using a finer-granularity TIOA program, which includes such notions as the input delay buffer and the individual processor steps and their time bounds, and proving that this lower-level program is a faithful implementation of the more abstract node TIOA program.

Acknowledgments: Josh Tauber contributed many ideas that appear in the discussions of the language, simulator and code generator. Anna Chefter worked on the detailed design and preliminary implementation of the simulator, and also helped with the coding of the front end and the data type implementation libraries. Mandana Vaziri designed a translation scheme to PROMELA and implemented a preliminary translator. Martin Rinard provided useful guidance and encouragement.

References

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

[2] INMOS Ltd: OCCAM 2 Reference Manual, 1988.

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

[4] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, March 1992.

[5] Myla Archer, 1997. Personal communication.

[6] Myla Archer and Constance Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In Oded Maler, editor, *Hybrid and Real-Time Systems* (International Workshop, HART'97, Grenoble, France, March 1997), volume 1201 of *Lecture Notes in Computer Science*, pages 171–185, Berlin Heidelberg, 1997. Springer-Verlag.

[7] Hagit Attiya and Nancy A. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. *Information and Computation*, 110(1):183–232, April 1994.

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

[9] A. Benviniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with event and relations: The SIGNAL language and its semantics. *Science of Computer Programming*, 16, 1991.

[10] Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm, December 1997. Submitted for journal publication.

[11] R. Braden. Extending TCP for transactions — concepts. Internet RFC-1379, July 1994.

[12] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Co., Reading, MA, 1988.

[13] S. Chaudhuri and P. Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. In *10th International Workshop on Distributed Algorithms*, October 1996. To appear in *Lecture Notes in Computer Science,* Springer Verlag.

[14] Anna E. Chefter. A simulator for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, May 1998.

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

[16] Gregory V. Chockler. An adaptive totally ordered multicast protocol that tolerates partitions. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, August 1997.

[17] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory - practical tools for specification, simulation, verification and implementation of concurrent systems. In *Specification of Parallel Algorithms. DIMACS Workshop.*, pages 75–89. American Mathematical Society, 1994.

[18] Roberto DePrisco. Revisiting the Paxos algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1997. Also, MIT/LCS/TR-717.

[19] Roberto DePrisco, Alan Fekete, Nancy Lynch, and Alex Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the 17th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 227–236, Puerto Vallarta, Mexico, June-July 1998. Also, technical memo in progress.

[20] Danny Dolev and Nir Shavit. Bounded time stamping. *SIAM Journal on Computing*, 26(2):418–455, April 1997.

[21] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55, Berlin, June 1992. Springer-Verlag. Proceedings of the Fourth International Conference, CAV'92.

[22] Zohar Manna et al. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Department of Computer Science, Stanford University, Stanford, CA 94305, June 1994.

[23] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data service. *Theoretical Computer Science on Distributed Algorithms (Special Issue)*. To appear.

[24] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 300–309, Philadelphia, PA, May 1996.

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

[26] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 53–62, Santa Barbara, CA, August 1997. Expanded version in [27].

[27] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. Technical Memo MIT-LCS-TM-570, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, 1997.

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

[29] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA a language for specifying, programming and validating distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 15 1997. URL `http://larch-www.lcs.mit.edu:8001/~garland/ioaLanguage.ps`.

[30] Rainer Gawlick, Nancy Lynch, and Nir Shavit. Concurrent time-stamping made simple. In D. Dolev, Z. Galil, and M. Rodeh, editors, *Theory of Computing and Systems* (ISTCS'92, Israel Symposium, Haifa, Israel, May 1992), volume 601 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, 1992.

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

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

[33] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, September 1995.

[34] M. J. C. Gordon. HOL: A proof generating system for higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 73–128. Springer-Verlag, 1989.

[35] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, October 1994.

[36] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing, editors. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, 1993.

[37] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[38] M. Hayden and R. van Renesse. Optimizing layered communication protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, November 1996.

[39] Mark Hayden. *Ensemble Reference Manual.* Cornell University, 1996.

[40] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[41] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for ensemble layers, May 1998. Manuscript in progress.

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

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

[44] Information Processing Systems Open Systems Interconnection ISO 9074. Estelle- a formal descrition technique based on a extended state transition model, 1989.

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

[46] Butler W. Lampson, Nancy A. Lynch, and Jørgen F. Søgaard-Andersen. Correctness of at-most-once message delivery protocols. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Formal Description Techniques, VI* (Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques — FORTE'93, Boston, MA, October 1993), pages 385–400. North-Holland, 1994.

[47] Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1995.

[48] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques* (FORTE'94, Berne, Switzerland, October 1994), pages 259–273. Chapman and Hall, 1995.

[49] Nancy Lynch. Modelling and verification of automated transit systems, using timed automata, invariants and simulations. Technical Memo MIT/LCS/TM-545, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, December 1995. Also, [51].

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

[51] Nancy Lynch. Modelling and verification of automated transit systems, using timed automata, invariants and simulations. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III: Verification and Control* (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 1995), volume 1066 of *Lecture Notes in Computer Science*, pages 449–463. Springer-Verlag, 1996. Also, [49].

[52] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[53] Nancy Lynch, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid I/O automata. Technical Memo MIT/LCS/TM-544, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, December 1995. Also, [54].

[54] Nancy Lynch, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid I/O automata. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III: Verification and Control* (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 1995), volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer-Verlag, 1996. Also, [53].

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

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

[57] Nancy Lynch and H. B. Weinberg. Proving correctness of a vehicle maneuver: Deceleration. In *Second European Workshop on Real-Time and Hybrid Systems*, pages 196–203, Grenoble, France, May/June 1995. Later version appears as [79].

[58] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, British Columbia, Canada, August 1987.

[59] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[60] Nancy A. Lynch and Frits W. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[61] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[62] José Meseguer. Conditional rewriting logic as an unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

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

[64] Tobias Nipkow. Formal verification of data type refinement: Theory and practice. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop, Mook, The Netherlands, May/June 1989), volume 430 of *Lecture Notes in Computer Science*, pages 561–591. Springer-Verlag, 1990.

[65] CoFI Task Group on Language Design. The common algebraic specification language, April 1998. URL http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/.

[66] Jonathan S. Ostroff. A visual toolset for the design of real-time discrete event systems. *IEEE Transactions on Control Systems Technology*, 5(3), May 1997.

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

[68] Tsvetomir P. Petrov, Anna Pogosyants, Stephen J. Garland, Victor Luchangco, and Nancy A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* (FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996), pages 29–44. Chapman & Hall, 1996.

[69] J. Postel. Transmission Control Protocol — DARPA Internet Program Specification (Internet Standard STC-007). Internet RFC-793, September 1981.

[70] N. Shankar, Sam Owre, and John Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab, SRI Intl., Menlo Park, CA, 1993.

[71] Mark Smith. Formal verification of communication protocols. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996, pages 129–144. Chapman & Hall, 1996.

[72] Mark Smith. *Formal Verification of TCP and T/TCP*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1997.

[73] Mark Smith. Reliable message delivery and conditionally-fast transactions are not possible without accurate clocks. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Distributed Computing*, pages 163–171, June 1998.

[74] Jørgen Søgaard-Andersen. *Correctness of Protocols in Distributed Systems*. PhD thesis, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, December 1993. ID-TR: 1993-131. Also, expanded version in MIT/LCS/TR-589.

[75] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th International Conference, CAV'93, Elounda, Greece, June/July 1993), volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[76] Ekrem Söylemez. Automatic verification of the timing properties of MMT automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, February 1994.

[77] M. G. Staskauskas. Formal derivation of concurrent programs: An example from industry. *IEEE Transactions on Software Engineering*, 19(5):503–528, May 1993.

[78] Joshua Tauber. IOA code generation - theory and practice. Manuscript.

[79] H. B. Weinberg and Nancy Lynch. Correctness of vehicle control systems: A case study. In *17th IEEE Real-Time Systems Symposium*, pages 62–72, Washington, D. C., December 1996. Earlier version appears as [57].