Simulating Nondeterministic Systems at Multiple Levels of Abstraction

Dilsun Kırlı Kaynar, Anna Chefter, Laura Dean, Stephen J. Garland
Nancy A. Lynch, Toh Ne Win, Antonio Ramírez-Robredo
MIT Laboratory for Computer Science*

Abstract

IOA is a high-level distributed programming language based on the formal I/O automaton model for asynchronous concurrent systems. A suite of software tools, called the IOA toolkit, has been designed and partially implemented to facilitate the analysis and verification of distributed systems using techniques supported by the formal model. An important proof technique for distributed systems defined by a hierarchy of abstractions involves the notion of a simulation relation between pairs of automata at different levels in the hierarchy. The IOA toolkit's simulator tests purported simulation relations by executing the low-level automaton and, given a proposed correspondence between its steps and those of the higher-level automaton, generating and checking an execution of the higher-level automaton. Once checked by the simulator, the simulation relation and the step correspondence can be used in conjunction with the toolkit's proof tools to construct a formal proof that the low-level automaton implements the higher-level one. This paper presents a case study that illustrates this use of the IOA toolkit to prove correct an algorithm for mutual exclusion. The case study shows how tools like the IOA simulator can play an important role in proving distributed systems correct.

1 Introduction

The input/output (I/O) automaton model [LT89, Lyn96] is a labeled transition system model suitable for describing systems with asynchronously interacting components. In this model, a component is represented as an I/O automaton which is a nondeterministic, possibly infinite-state, state machine. The external behavior of each automaton is defined by a simple mathematical object called a trace.

The I/O automata model supports viewing systems at multiple levels of abstraction. A system can be described first at a high-level of abstraction, capturing only the essential requirements about its behavior, and then be refined successively until the desired level of detail is reached. The model defines what it means for an automaton to implement another (in terms of trace inclusion), and it introduces the notion of a simulation relation as a sufficient condition to prove an implementation relation between two automata. A parallel composition operator, also included in the model, allows one to decompose the description, analysis and verification of large and complex systems.

IOA [GL00, GL98] is a formal language for describing I/O automata. It can be regarded as a high-level distributed programming language. Its design was driven by a motivation to support both simulation [Che98, RR00, Dea01] and verification [Bog01]. The IOA toolkit is a partially

*Corresponding address: 200 Technology Square, Cambridge, MA 02139, USA, dilsun@theory.lcs.mit.edu. Currently, Chefter is employed by Merill Lynch, Dean is employed by Oryxa, and Ramírez is in the PhD program in mathematics at Stanford University.

implemented set of software tools that support the design, analysis, and development of systems within the I/O automaton framework. The toolkit contains a front-end that checks whether system descriptions (IOA programs) comply with IOA's syntax and static semantics, and that produces an intermediate representation of the code for use by the back-end tools (a simulator, interfaces to a number of existing theorem provers, model checkers, and an automatic code generator).

A key feature of the I/O automaton model is nondeterminism. Nondeterminism allows systems to be described in their most general forms and to be verified considering all possible behaviors without being tied to a particular implementation of a system design. The results obtained for a nondeterministic system carry over to different implementations of the same system. Nondeterminism also makes it easier to prove correctness in the absence of extraneous, unnecessary restrictions. A key challenge in the design of IOA has been to provide support for both simulation and verification in a unified framework. Nondeterminism in IOA assists verification in the ways noted above. On the other hand, nondeterminism complicates simulation, which must choose particular executions. Therefore, simulation requires mechanisms for resolving nondeterminism. The IOA language and toolkit provide such mechanisms. Moreover, these mechanisms turn out to be useful not just for simulation, but for verification as well.

In this paper, we describe by means of a case study how the IOA toolkit can be used for simulating and subsequently verifying distributed algorithms. We focus on the capability of the IOA simulator to simulate pairs of I/O automata at different levels of abstraction. Users present the paired simulator with descriptions of two automata, a candidate simulation relation, and a mapping, called a step correspondence, from the actions of the lower-level automaton to sequences of actions of the higher-level one. The simulator simulates the low-level automaton, checks whether the trace of the high-level automaton induced by the step correspondence is identical to that of the low-level automaton, and checks whether the candidate simulation relation holds throughout the simulated executions.

In our case study we present an algorithm for mutual exclusion and use the paired simulator to obtain evidence that this algorithm satisfies the mutual exclusion property. We then verify that the algorithm satisfies this property with LP [GG91]. The toolkit facilitates the automatic translation of the algorithm and the candidate simulation relation into the language of LP.

Related work Other toolkits such as AsmL [GSV01] tools, Mocha [dAAG$^+$00], the SMV system [McM], and TLC [LY01] support simulation or verification of concurrent and distributed systems. The IOA toolkit differs from these in that it combines paired simulation capability with theorem-proving based verification. AsmL facilitates simulating systems at different levels of abstraction, checking step by step whether a system satisfies its specification, but it does not support using paired simulation in conjunction with proof tools. The verification components of Mocha, SMV, and TLC use model checking and hence are limited to exploring finite state spaces; the proof tools in the IOA toolkit apply to finite and infinite systems alike. Another feature that distinguishes the IOA toolkit from other tools is the connection of its simulator to a program analysis tool [ECGN01] for automatic invariant discovery.

2 I/O automata and the IOA Language

This section includes a brief introduction to the I/O automaton model and the IOA Language. We refer the reader to [Lyn96, GL98] for an in-depth introduction.

2.1 Theoretical background

An I/O automaton is a simple type of state machine in which the transitions between states are associated with named *actions* π. The actions are classified as either *input, output*, or *internal.* The input actions are assumed not to be under the automaton's control, whereas the automaton itself controls which output and internal actions should be performed. An I/O automaton consists of a *signature*, which lists its actions, a set of *states*, some of which are distinguished as start states, a *state-transition relation*, which contains triples of the form (state, action, state), and an optional set of *tasks.* We do not consider automata with tasks in this paper.

An action π is said to be enabled in a state s if there is another state s' such that (s, π, s') is a transition of the automaton. Input actions are enabled in every state. The operation of an I/O automaton is described by its *executions* $s_0, \pi_1, s_1, \ldots$, which are alternating sequences of states and actions, and its *traces*, which are the externally visible behavior occurring in executions. One automaton is said to *implement* another if all its traces are also traces of the other. The *parallel composition* operator allows an output action of one automaton to be identified with input actions in other automata; this operator respects the trace semantics.

The I/O automaton model provides support for system descriptions at multiple levels of abstraction. The process moving through a series of abstractions, from higher to lower levels, is called *successive refinement.* To prove that one automaton implements another, one needs to show that for any execution of the lower level automaton there is a corresponding execution of the higher level automaton. The notion of a *simulation relation* proves useful in constructing proofs of implementation relations.

Definition 2.1 (Forward simulation). A forward simulation from automaton A to automaton B is a relation f on $states(A) \times states(B)$ with the following properties:

1. For every start state a of A, there exists a start state b of B such that $f(a, b)$.
2. If a is a reachable state of A, b is a reachable state of B such that $f(a, b)$, and $a \xrightarrow{\pi} a'$, then there exists a state b' of B and an execution fragment β of B such that $b \xrightarrow{\beta} b'$, $f(a', b')$ holds, and $trace(\pi) = trace(\beta)$.

Theorem 2.1. If there is a forward simulation relation from A to B, then every trace of A is a trace of B. (See [Lyn96] for a proof.)

2.2 The IOA language

In the IOA language, the description of an I/O automaton has four main parts: the action signature, the states, the transitions, and the tasks of the automaton. States are represented by collections of typed variables. The transition relation is usually given in precondition-effect style, which groups together all transitions that involve a particular action into a single piece of code. Each definition has a precondition (indicated by the keyword **pre**), which describes a condition on the state that should be true before the transition can be executed, and an effect (indicated by the keyword **eff**) which describes how the state changes when the transition is executed. The entire piece of code in the effect of a transition is executed indivisibly. If **pre** is not specified, then it is assumed to always hold.

The code may be written either in an imperative style, as a sequence of assignment, conditional, and looping instructions, or in declarative style, as a predicate relating state variables in the pre- and post-states, transition parameters, and nondeterministic parameters. It is also possible to use a combination of these two styles.

Nondeterminism appears in IOA in two ways: *explicitly* in the form of **choose** constructs in state variable initializations and the effects of the transition definitions, and *implicitly*, in the form of action scheduling uncertainty. We present examples for both forms of nondeterminism later in the paper and describe how they are resolved by the IOA simulator.

2.3 Example: Specification of mutual exclusion

We present a sample IOA program to illustrate some of the language constructs discussed above and to introduce the mutual exclusion problem that constitutes the basis of our case study. We build on this example gradually as we discuss simulation and proof techniques based on simulation relations.

The mutual exclusion problem involves the allocation of a single, indivisible, non-shareable resource among n processes. The resource could be an output device that requires exclusive access to produce sensible output or a data structure that requires exclusive access in order to avoid interference among the operations of different processes. A process with access to the resource is modeled as being in a *critical region*, which is a designated subset of its states. When a process is not involved in any way with the resource, it is said to be in the *remainder region*. In order to gain admittance to its critical region, a process executes a *trying protocol*; after it is done with the resource, it executes an *exit protocol*. This procedure can be repeated so that each process follows a cycle, moving from its remainder region to its trying region and arriving back at the remainder region after going through critical and exit regions.

We consider mutual exclusion within the shared memory model explained in [Lyn96]. The shared memory system contains n processes, numbered $1, \ldots, n$. The $try, crit, exit$, and rem actions are the only external actions of a process. Input actions consist of try_i, which models a request for access to the resource by process i, and $exit_i$, which models an announcement that process i is done with the resource. Output actions consist of $crit_i$, which models the granting of access to process i, and rem_i, which tells process i that it can continue with the remainder of its work. Formally, we define a sequence of $try_i, crit_i, exit_i$, and rem_i actions to be well-formed for process i if it is a prefix of the cyclically ordered sequence $try_i, crit_i, exit_i, rem_i, try_{i'}, \ldots$ The automaton `Mutex` (Figure 1) is an IOA specification of mutual exclusion for three processes in which the well-formedness of interaction with the environment is guaranteed.

The state variable `regionMap` maps process indices to regions and keeps track of the current region of each process. The initialization of `regionMap` to `constant(rem)` defines the start state. The transition definitions are mostly self-explanatory. Each action updates the variable `regionMap` to record the region entered upon its execution. The transition definition for `crit` imposes the mutual exclusion condition: a process in a trying region is allowed to enter its critical region only if there is no other process that is in region `crit`.

2.4 Example: An algorithm for mutual exclusion

Figure 2 contains IOA code for an algorithm for mutual exclusion. Comments in the code indicate items that will be of particular interest when we discuss the mechanism for resolving nondeterminism to enable simulation. We start, however, by explaining the algorithm briefly, pointing at the sources of nondeterminism.

The algorithm described by the automaton `DijkstraInt` is a simplified version of a mutual exclusion algorithm by Dijkstra presented in [Lyn96]. It abstracts away those parts in the original algorithm dedicated to dealing with liveness. The suffix "Int" in the automaton name indicates that we consider it to be an intermediate level algorithm: not at as high a level as the specification,

```
type Index = enumeration of p1, p2, p3
type Region = enumeration of rem, try, crit, exit

automaton Mutex
  signature output try(p: Index), crit(p: Index), exit(p: Index), rem(p: Index)
  states regionMap: Array[Index, Region] := constant(rem)
  transitions
    output try(p)
      pre regionMap[p] = rem
      eff regionMap[p] := try
    output crit(p)
      pre regionMap[p] = try ∧ ∀ u: Index (p ≠ u ⇒ regionMap[u] ≠ crit)
      eff regionMap[p] := crit
    output exit(p)
      pre regionMap[p] = crit
      eff regionMap[p] := exit
    output rem(p)
      pre regionMap[p] = exit
      eff regionMap[p] := rem
```

Figure 1: Specification of mutual exclusion

yet less detailed than the original algorithm of Dijkstra.

The automaton `DijkstraInt` uses two types, `PcValue` and `Stage`, in addition to those in Figure 1. Values of type `PcValue` represent possible program counter values for a process, while values of type `Stage` represent stages of the algorithm. The automaton has four external and four internal actions. The external actions have the same names as those of `Mutex` in Figure 1. This is no coincidence, as our ultimate aim is to show that `DijkstraInt` implements mutual exclusion as specified by `Mutex`.

The algorithm has two stages. The first, `stage1`, indicates that a process is either inactive or is about to enter the second stage. The second, `stage2`, embodies the crucial steps and determines whether a process is allowed to enter its critical region. A process can enter its critical region only if all other processes are in `stage1`. The transition definition for action `check` details how this works. Each process p uses a set $S[p]$ to keep track of the processes that it has detected as being in `stage1`. The state variables `flag` and `pc` record the stage of the algorithm for each process and control the order of occurrence of the actions mimicking the program counter for a process.

Explicit nondeterminism in this example arises from the **choose** statement in the transition definition for action `check`. When a process p performs the `check` action, it nondeterministically chooses the process u to be checked. The predicate in the **where** clause allows the nondeterministic choice to yield any process that is not already in the set $S[p]$. Implicit nondeterminism also arises in this example, because there may be more than one action enabled at a time. Consider, for example, the very first action to be performed by the automaton. Since the program counters (`pc`) of all processes are initialized to `rem`, all processes are enabled to perform the `try` action. To simulate this automaton, one must select one of these processes to start execution.

3 Simulation and nondeterminism resolution

The simulator runs sample executions of an IOA program, allowing the user to help select the executions. It generates logs of execution traces and displays information upon the user's request. The IOA Language allows users to propose invariants, which the simulator checks in the selected executions.

The simulator requires that IOA programs be transformed into a form suitable for simulation.

```
type PcValue = enumeration of rem, setflag1, setflag2, check, leavetry,
                               crit, reset, leaveexit
type Stage = enumeration of stage1, stage2

automaton DijkstraInt
  signature
    output try(p: Index), crit(p: Index), exit(p: Index), rem(p: Index)
    internal setflag1(p: Index), setflag2(p: Index), check(p: Index), reset(p: Index)
  states
    flag: Array[Index, Stage] := constant(stage1),
    pc: Array[Index, PcValue] := constant(rem),
    S: Array[Index, Set[Index]] := constant({}),
    u: Index
  transitions
    output try(p)
      pre pc[p] = rem
      eff pc[p] := setflag1
    internal setflag1(p)
      pre pc[p] = setflag1
      eff flag[p] := stage1; pc[p] := setflag2
    internal setflag2(p)
      pre pc[p] = setflag2
      eff flag[p] := stage2; S[p] := {p}; pc[p] := check
    internal check(p)
      pre pc[p] = check
      eff u := choose x: Index where ¬(x ∈ S[p]);
                 %% explicit nondeterminism to be resolved for simulation
          if flag[u] = stage2 then S[p] := {}; pc[p] := setflag1
          else S[p] := S[p] ∪ {u};
               if ∀ i: Index (i ∈ S[p]) then pc[p] := leavetry  fi
          fi
    output crit(p)
      pre pc[p] = leavetry
      eff pc[p] := crit
    output exit(p)
      pre pc[p] = crit
      eff pc[p] := reset
    internal reset(p)
      pre pc[p] = reset
      eff flag[p] := stage1; S[p] := {}; pc[p] := leaveexit
    output rem(p)
      pre pc[p] = leaveexit
      eff pc[p] := rem
    %% implicit nondeterminism to be resolved for simulation
```

Figure 2: An algorithm for mutual exclusion

The crucial problem in this transformation is resolving nondeterminism. The nondeterminism resolution approach adopted by the IOA simulator is to assign a program, called an *NDR program*, to each source of nondeterminism in an automaton. There is an NDR program corresponding to every **choose** statement, and an NDR program for scheduling the actions of the automaton. We explain the nondeterminism resolution mechanism of the IOA simulator by referring to the example presented in Section 2.4.

3.1 Resolving explicit nondeterminism

A simple NDR program (**det**erminator), given below, resolves the explicit nondeterminism for the `check` action in the automaton `DijkstraInt`. It **yield**s a process index that is not in $S[p]$. This index is guaranteed to differ from p because p is placed in $S[p]$ before `check` is enabled, and it is guaranteed to exist because `check` is no longer enabled once $S[p]$ contains all indices.

```
det do
       if ¬(p1 ∈ S[p]) then yield p1
       elseif ¬(p2 ∈ S[p]) then yield p2
       elseif ¬(p3 ∈ S[p]) then yield p3
       fi
    od
```

3.2 Resolving implicit nondeterminism

To resolve implicit nondeterminism, users of the IOA simulator must specify a scheduling policy using the language constructs of IOA. We present below a sample schedule block that implements a randomized scheduling policy for three processes. It picks a random integer between 1 and 3 and uses this integer to decide which process will be given the turn to perform an action. It checks the enabling conditions for the randomly chosen process and **fire**s the enabled action. The **while** loop that contains these steps is nonterminating; the IOA simulator prompts the users for the maximum number of steps to simulate and halts the execution automatically when the predetermined step is reached.

```
schedule
  states pick: Int, p: Index
  do while true do
        pick:= randomInt(1,3);
        if pick = 1 then p := p1
        elseif  pick = 2 then p := p2
        else p := p3
        fi;
        if pc[p] = rem  then fire output try(p)
        elseif pc[p] = setflag1 then fire internal setflag1(p)
        elseif pc[p] = setflag2 then fire internal setflag2(p)
        elseif pc[p] = check then fire internal check(p)
        elseif pc[p] = leavetry then fire output crit(p)
        elseif pc[p] = crit then fire output exit(p)
        elseif pc[p] = reset then fire internal reset(p)
        else  fire output rem(p)
        fi
     od
 od
```

3.3 Checking invariants

The IOA simulator checks the validity of invariants proposed by users. We present below several invariants for the automaton `DijkstraInt` that are key lemmas for proving the algorithm correct.

In Section 5 we take up the question of how the user discovers such lemmas.

Each process p uses a set $S[p]$ to keep track of successfully checked processes, that is, of processes that are not contending with p to enter the critical region. The first assertion states that two processes cannot both be executing the second stage of the algorithm and be in each other's set. The second states that whenever the `pc` value for a process is `leavetry` or `crit`, its set contains all of the processes. These two assertions express the key ideas we will use in our proof: if the `pc` values for two processes were `crit` at the same time, it would be impossible for `assertion1` and `assertion2` to be both true.

```
invariant assertion1 of DijkstraInt:
  ∀ i: Index ∀ j: Index ¬(i ≠ j ∧ flag[i] = stage2 ∧ flag[j] = stage2
                           ∧ i ∈ S[j] ∧ j ∈ S[i])

invariant assertion2 of DijkstraInt:
  ∀ i: Index (   (pc[i] = leavetry ⇒ ∀ j:Index (j ∈ S[i]))
              ∧ (pc[i] = crit     ⇒ ∀ j:Index (j ∈ S[i])) )
```

3.4 Simulator output

The automaton `DijkstraInt` from Figure 2 can be simulated with the IOA simulator after inserting the NDR programs specified in Section 3 in the indicated places. The invariants to be checked need to be appended to the code.

Some output of the simulator for running `DijkstraInt` is shown below. It displays the step involving the first entry to the critical region (step 21) in a simulation for 200 steps. The simulator reports errors if any of the invariants fail at a simulated step, if an NDR program attempts to fire a transition that is not enabled, or if it attempts to yield a value that does not satisfy the **where** clause of the corresponding **choose** statement.

```
[[[[ Begin step 21 [[[[
     transition: output crit(p1) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 rem))
]]]] End step 21 ]]]]
```

4 Paired Simulation

In this section, we describe how the simulator simulates execution of a pair of automata related by a simulation relation as defined in Section 2. The key problem here is that simulation relation, being merely a predicate that relates the states of two automata, does not identify how each step in the implementation automaton corresponds to a sequence of steps in the specification automaton. In general, there might be multiple step correspondences that realize a given valid simulation relation between automata; even if there is only one, it can be difficult to find it. The problem of deriving a specification-level execution from an implementation-level execution is analogous to that of deriving a deterministic execution of a single automaton from a specification that allows nondeterminism.

The design of the paired simulator is based on the observation that it is reasonable and beneficial to require users to specify a step correspondence. In most correctness proofs, determining when a particular action in the specification is performed by the implementation turns out to be the key to the proof. By requiring a user to specify the step correspondence, the simulator actually urges the user to understand the relationship between the two levels. Once the main invariants and the step correspondence is determined, the rest of the proof is likely to involve routine bookkeeping steps.

4.1 Encoding step correspondences

A step correspondence needs to specify, for a given low-level transition, a high-level execution fragment such that execution of both the low-level transition and the high-level fragment preserves the simulation relation. Thus, a step correspondence can be seen as an "attempted proof" of the simulation relation, missing only the reasoning that shows that the simulation relation is preserved. To specify the proposed proof of a simulation relation, the IOA **forward simulation** assertion allows a section called **proof** for specifying the step correspondence. This section contains one entry for each possible transition definition in the low-level automaton; each entry provides an algorithm for producing a high-level execution fragment. In addition to these entries, the **proof** section contains an initialization block, which specifies how to set the variables of the high-level automaton given the initial state of the low-level automaton, and an optional **states** section that declares auxiliary variables used by the step correspondence.

4.2 Example: Forward simulation from `DijkstraInt` to `Mutex`

Figure 3 defines a forward simulation relation in IOA and contains a proof block for that relation. Together with the IOA descriptions of `Mutex` and `DijkstraInt` augmented with the NDR programs from Section 3, this block allows one to use the paired simulator to check whether the relation holds in the simulated executions.

The candidate relation in this example is based on the relation between the values of the state variable `pc` of the low-level automaton and those of the state variable `regionMap` of the specification automaton. The intuition behind this relation is as follows. For each region in the specification of mutual exclusion there are certain actions that can be performed by the low-level automaton. These actions are determined by the `pc` values. The relation states that whenever the program counter of a process at the low-level automaton is set to one of `setflag1`, `setflag2`, `check`, or `leavetry`, the `regionMap` of the specification automaton must show region `try` for the same process. The rest of the relation is defined similarly. The delimiter ";" can be interpreted as conjunction.

In paired simulation, the simulation of the low-level algorithm drives the simulation of the high-level one. For each external action performed by the low-level automaton, the proof block directs the simulator to fire the action with the specified name at the high-level. The internal actions are matched by empty execution fragments indicated by **ignore** statements. The simulator checks whether the proposed simulation relation holds after the actions are performed. The following is a sample output of the paired simulator, displaying the simulation step 17.

```
[[[[ Begin step 17 [[[[
     Executed impl transition: output crit(p1) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 setflag2))
     Executed spec transition: output crit(p1) in automaton Mutex
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p1 crit) (p2 try) (p3 try))
]]]] End step 17 ]]]]
```

Note that the simulator gives information about how the states of the two automata change upon the occurrence of an action of the implementation automaton. In this example, each step in the low-level execution is matched by either a single step or an empty execution fragment in the specification. The IOA simulator can also handle paired simulations in which this is not the case. It allows execution fragments to be specified by any IOA program consisting of assignments, conditional, while, and fire statements. For example, a step correspondence in which an output

```
forward simulation from DijkstraInt to Mutex :
  ∀ i: Index (DijkstraInt.pc[i] = setflag1 ∨ DijkstraInt.pc[i] = setflag2 ∨
              DijkstraInt.pc[i] = check ∨ DijkstraInt.pc[i] = leavetry
                ⇔ Mutex.regionMap[i] = try);
  ∀ i: Index (DijkstraInt.pc[i] = crit ⇔ Mutex.regionMap[i] = crit);
  ∀ i: Index (DijkstraInt.pc[i] = rem ⇔ Mutex.regionMap[i] = rem);
  ∀ i: Index (DijkstraInt.pc[i] = reset ∨ DijkstraInt.pc[i] = leaveexit
                ⇔ Mutex.regionMap[i] = exit);
proof
    initially Mutex.regionMap := constant(rem)
    for output try(p:Index) do fire output try(p) od
    for output crit(p:Index) do fire output crit(p) od
    for output exit(p:Index) do fire output exit(p) od
    for output rem(p:Index) do fire output rem(p) od
    for internal setflag1(p:Index) ignore
    for internal setflag2(p:Index) ignore
    for internal check(p:Index) ignore
    for internal reset(p:Index) ignore
```

Figure 3: Forward simulation from `DijkstraInt` to `Mutex`

action `a` at the low-level is matched by a sequence consisting of an output action `a` that is preceded and followed by an internal action `b` could be encoded as follows:

```
     for output a do fire internal b; fire output a; fire internal b od
```

5 Using simulation results to help construct a proof of correctness

In the previous section we introduced a method for simulating pairs of automata at different levels of abstraction with the aid of the IOA toolkit. It is important to note that paired simulation provides only empirical evidence for the correctness of a simulation relation. In most cases it is desirable to complement this evidence with a proof. In this section we describe the support provided by the IOA toolkit for formal verification.

5.1 Method

The IOA toolkit has been designed to support verification of *safety properties*, which specify that a "bad' event never happens. LP is an interactive theorem proving system for multisorted first-order logic and is suitable for reasoning about safety properties expressible in this kind of logic. It admits specifications of theories in the Larch Shared Language (LSL). The IOA toolkit includes a tool called `ioa2lsl` [Bog01], which translates IOA definitions of automata, their invariants, and simulation relations into LSL theories. The tool `ioa2lsl` combines the definition of an automaton with standard LSL definitions of I/O automata to produce axioms in first-order logic that describe the operation of the automaton. These are subsequently used to generate input for LP.

5.2 Example: Proof of forward simulation

We now describe how we proved that a candidate simulation relation, presented in Figure 3 and checked with the paired simulator for selected executions, is actually a forward simulation relation from `DijkstraInt` to `Mutex`. It then follows from Theorem 2.1 that `DijkstraInt` implements mutual exclusion.

We first used `ioa2lsl` to process the file containing the definitions of the two automata, their invariants, and the simulation relation.[1] We then used the LSL Checker to prepare the axioms and proof obligations for LP.

The proof of the simulation relation proceeds by induction. The basis step consists of showing that the relation holds for the start state. The proof of the induction step takes the form of proof by cases. The heart of the proof lies in providing a "witness" for an existential quantifier asserting the existence of a simulating step sequence in the high-level automaton that preserves the simulation relation and has the the same trace as a given step of the low-level automaton. The step sequence already constructed for the paired simulator turns out to be exactly what is needed to provide this witness.

Proofs of the invariants were routine proofs by induction. The proof of `assertion1` gave rise to the need to prove two other simpler invariants:

```
invariant assertion3 of DijkstraInt:
   ∀ i: Index (pc[i] = leavetry ⇒ flag[i] = stage2)

invariant assertion4 of DijkstraInt:
   ∀ i: Index (pc[i] = crit ⇒ flag[i] = stage2)
```

5.3 Automatic detection of invariants

Finding key invariants is an essential step in proofs of correctness. Any help from automatic tools in finding these invariants would alleviate the burden on the user. For example, if a tool could discover simple invariants such as `assertion3` and `assertion4`, which LP can prove more or less automatically, and if LP could use these to prove the invariants `assertion1` and `assertion2` used in the correctness proof, that proof would become much easier.

We have begun [WE02] developing this kind of automated proof assistance by connecting the IOA simulator to Daikon [ECGN01], a tool for dynamic invariant discovery. The user can instruct the IOA simulator to record the values of state variables upon entry to and exit from each transition in the course of a selected execution. Then Daikon can infer invariants about the pre-state and post-state of each transition by examining these values.

In our preliminary experiments, Daikon was able to infer some potentially useful invariants. For example, Daikon detected that `flag[p]=stage2` in the pre-state of `crit(p)`. The invariant `assertion3` in the previous is just the implication of this invariant by the precondition of the `crit` action. We are continuing to work on the Daikon-IOA connection to detect other useful invariants and to automate the formulation of invariants such as `assertion3`.

6 Overview of the implementation

A preliminary "IOA toolkit distribution" (software package including source and Java executables) is available from the home page of the IOA project (`http://theory.lcs.mit.edu/tds/ioa.html`).

The front-end of the toolkit takes IOA descriptions and LSL specifications as input and outputs an equivalent specification written in an intermediate language. Each back-end tool takes as input the intermediate form of an IOA specification. There is common support for the back-end tools in the form of an intermediate language parser and an internal representation of IOA elements, in the form of a Java class hierarchy.

Data types are defined axiomatically in IOA so as to facilitate their translation into theorem prover input languages. We provide definitions for built-in data types and allow the programmer to

[1]The tool `ioa2lsl` is still under development, and we had to edit its output to correct a number of small errors.

define new data types using LSL. However, in order to simulate data type operations, the simulator needs actual code for the specified operations. Each IOA sort is implemented by a Java class, and each operator is implemented by a method on that class. The implementation classes extend the `ioa.runtime.ADT` class, which provides two operators common to all IOA data types. The simulator obtains implementations for sorts and operators by querying a global *implementation registry.*

The simulator shares runtime type libraries with the IOA code generator to ensure similar code behavior and to reduce repeated code [Tsa02].

7 Discussion and conclusions

Formal correctness proofs for distributed systems can be long, hard, or tedious to construct. Simulation can be used as a way of testing system designs before delving into correctness proofs. It either reveals bugs or increases confidence that a system behaves as expected. Simulation can also assist users in constructing correctness proofs. It is this aspect of simulation that we focused on throughout this paper.

We considered nondeterministic systems modeled using the I/O automata formalism and described how these systems can be simulated with the support of the IOA language and the toolkit. Our aim was to draw attention to a useful capability of the IOA simulator – paired simulation – that allows users to check whether two automata at different levels of abstraction are related by a simulation relation for the selected executions. In the I/O automaton model, the notion of a simulation relation between two automata is a useful conceptual tool to prove the correctness of systems. Hence, the ability to propose and check simulation relations with the IOA simulator constitutes a valuable step towards a formal proof based on a simulation relation. The specification of a relation is not the only thing that is required from a user by the paired simulator. A user is also required to specify a step correspondence that will make the simulation relation hold throughout paired simulation. This is particularly useful since finding the right step correspondence is usually the key to the proof of a simulation relation. This indeed happened in our case study.

Another capability of the IOA simulator that helps the construction of proofs is invariant checking. The invariants that are observed to be true for simulated executions constitute candidates for useful lemmas. The invariants that we checked with the paired simulator in our case study were later used as lemmas in the full proof.

The case study in this paper suggests a general methodology for the analysis and verification of distributed systems with the IOA toolkit, using multiple levels of abstraction. The basic steps are to:

1. Write the IOA code for the specification and the implementation automata;
2. For each automaton, resolve nondeterminism and perform simulation to test that the automaton behaves as expected;
3. Formulate a candidate forward simulation relation from the implementation automaton to the specification automaton, specify a step correspondence and perform paired simulation to check whether the relation holds for the selected executions;
4. Formulate the potentially useful invariants for the proof of the simulation relation and check whether they are true for the selected executions;
5. Use the tool ioa2lsl to translate the IOA code for automata and the forward simulation relation to LSL, and to generate proof obligations for LP; and

6. Prove with LP that the simulation relation holds for all possible executions, making use of the step correspondence and the key invariants.

A current project aims at improving the connection between the program analysis tool Daikon and the IOA simulator. We expect this connection to contribute to this methodology by automating parts of the correctness proofs.

References

[Bog01] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2001.

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

[dAAG$^+$00] L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang. *Mocha: Exploiting Modularity in Model Checking.* University of California at Berkeley Department of Electrical Engineering and Computer Sciences, University of Pennsylvania Department of Computer and Information Sciences, 2000. URL `http://www-cad.eecs.berkeley.edu/~mocha/refs.shtml`.

[Dea01] Laura G. Dean. Improved simulation of Input/Output automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2001.

[ECGN01] Michael Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[GG91] Stephen Garland and John Guttag. *A guide to LP, the Larch Prover.* Technical report, DEC Systems Reserach Center, 1991. Updated version avaliable at URL `http://nms.lcs.mit.edu/Larch/LP`.

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

[GL00] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, USA, 2000.

[GSV01] Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, Microsoft Research, 2001. URL for software `http://www.research.microsoft.com/foundations/asml/`.

[LT89] 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.

[LY01] Leslie Lamport and Yuan Yu. *TLC – The TLA+ Model Checker.* Compaq Systems Research Center, Palo Alto, California, 2001. URL `http://research.microsoft.com/users/lamport/tla/tlc.html`.

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

[McM] K. L McMillan. *The SMV Language.* Cadence Berkeley Labs, 2001 Addison Street, Berkeley, CA 94 704, USA. URL `http://www.cis.ksu.edu/santos/smv-doc/`.

[RR00] J. Antonio Ramirez-Robredo. Paired simulation of I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2000.

[Tsa02] Michael Tsai. Code generation for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 2002.

[WE02] Toh Ne Win and Michael Ernst. Verifying distibuted algorithms via dynamic analysis and theorem proving. Technical Report MIT-LCS-TR-841, MIT Laboratory for Computer Science, May 2002.

A Proof Script

The following is the Larch proof script for the simulation relation presented in Section 4. The set of axioms `DijkstraInt2Mutex_Axioms` is generated by the LSL checker by processing the LSL specification of the simulation relation. The tactics that are referred to in the proof are given in Section A.2.

A.1 Main simulation proof

```
execute DijkstraInt2Mutex_Axioms

declare variables u': States[DijkstraInt], act: Actions[DijkstraInt], pi: ActionSeq[Mutex]


set name theorem
prove start(u:States[DijkstraInt]) => \E s:States[Mutex] (start(s:States[Mutex]) /\ F(u, s))
  resume by specializing s:States[Mutex] to [constant(rem)]
    execute tactic_implies
qed

prove
  isStep(u:States[DijkstraInt], act, u') /\ F(u, s)
  /\ assertion1(u) /\ assertion2(u) /\ assertion3(u) /\ assertion4(u)
    => \E pi:ActionSeq[Mutex] (execFrag(s, pi) /\ trace(pi:ActionSeq[Mutex]) = trace(act)
                               /\ first(s, pi) = s /\ F(u', last(s, pi)))
  ..
  resume by induction on act
    % try action
    resume by =>
      resume by specializing pi to try(i1c) * {}
        execute tactic_and4cases
    % crit action
    resume by =>
      resume by specializing pi to crit(i1c) * {}
        resume by /\
          critical-pairs *Hyp with *Hyp
          execute tactic_4cases
          resume by =>
            resume by contradiction
              critical-pairs *Hyp with *Hyp
    % exit action
    resume by =>
```

```
      resume by specializing pi to exit(i1c) * {}
        execute tactic_and4cases
    % rem action
    resume by =>
      resume by specializing pi to rem(i1c) * {}
        resume by /\
          critical-pairs *Hyp with *Hyp
          execute tactic_4cases
    % setflag1 action
    resume by =>
      resume by specializing pi to {}
        critical-pairs *Hyp with *Hyp
        execute tactic_and4cases
    % setflag2 action
    resume by =>
      resume by specializing pi to {}
        critical-pairs *Hyp with *Hyp
        execute tactic_and4cases
    % check action
    resume by =>
      resume by specializing pi to {}
        critical-pairs *Hyp with *Hyp
        resume by /\
          execute tactic_stage2_i2c
          execute tactic_stage2_i2c
          execute tactic_stage2_i2c
          execute tactic_stage2_i2c
    % reset action
    resume by =>
      resume by specializing pi to {}
        critical-pairs *Hyp with *Hyp
        exe tactic_and4cases
qed
quit
```

A.2 Tactics

```
    % tactic_implies
    resume by =>

    % tactic_case
    res by case i1c = i

    % tactic_4cases
    execute tactic_case
    execute tactic_case
    execute tactic_case
    execute tactic_case

    % tactic_and4cases
    resume by /\
          execute tactic_4cases

    % tactic_stage2_i2c.lp
    resume by case uc.flag[i2c] = stage2
            execute tactic_case
            resume by case \A i:Index (i = i2c \/ i \in uc.S[i1c])
              execute tactic_case
```