The IOA Simulator [1]

Dilsun Kırlı Kaynar, Anna Chefter, Laura Dean,
Stephen Garland, Nancy Lynch, Toh Ne Win, Antonio Ramírez-Robredo
MIT Laboratory for Computer Science [2]

July 16, 2002

[1]Available at URL http://theory.lcs.mit.edu/tds/ioa.html.

[2]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.

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 systems using techniques supported by the formal model. This paper introduces the IOA simulator [1] which is a part of the IOA toolkit.

The IOA simulator runs selected executions of an I/O automaton on a single machine, generates logs of execution traces and displays information about the selected executions. The simulator also has the capability to simulate pairs of I/O automata, allowing users to check purported simulation relations between automata described at different levels of abstraction.

This paper is a primary source of reference for both the users and the developers of the IOA simulator. It describes the design of the simulator focusing on the mechanism for resolving nondeterminism in IOA programs. It includes a collection of small examples to illustrate the basic concepts regarding the simulation of IOA programs, and a larger tutorial example that demonstrates how to use the simulator. The final section of the paper gives information about the implementation of the simulator.

Acknowledgements This work was funded by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, by DARPA through the Office of Naval Research under contract number N66001-99-2-891702, by NTT under contract number MIT9904-12, by NSF under contract number ACI-9876931 and by AFOSR-ONR under contract number F49620-94-1-0199.

[1] The instructions for obtaining the related software can be found at `http://theory.lcs.mit.edu/tds/ioa.html`.

Contents

1 Introduction

1.1 Overview

The development of formal methods for modeling and reasoning about distributed systems is one of the major research activities within the Theory of Distributed Systems Group at MIT. The input/output automaton (I/O automaton) model [LT89, Lyn96] constitutes the basis of the work on formal methods. It is a labeled transition system model suitable for describing asynchronous concurrent systems [Lyn96].

The I/O automaton model incorporates the notion of abstraction to enable viewing systems at multiple levels of abstraction. A system can be first described at a high level of abstraction, capturing only the essential requirements about its behavior, and then be successively refined until the desired level of detail is reached. The model defines what it means for an automaton to implement another and introduces the notion of a simulation relation as a sufficient condition to prove an implementation relation between two automata.

The notion of parallel composition, also included in the I/O automaton model, facilitates modular design and analysis of distributed systems. The parallel composition operator in the model allows one to construct large and complex systems from smaller and simpler subsystems and study their behavior in terms of the behaviors of its components.

Work on the I/O automaton model includes the definition of a formal language—the IOA language [GL00, GL98]—for describing I/O automata. The IOA language can be regarded as a high-level distributed programming language. Its design has been driven by the motivation to support both simulation and verification. A suite of software tools—the IOA toolkit—is being developed to facilitate the design, analysis, and development of systems within the I/O automaton framework. The toolkit consists of a front-end that checks whether system descriptions (IOA programs) comply with the IOA syntax and static semantics, and produces an intermediate representation of the code to be used by the back-end tools. The back-end tools include the IOA simulator, a code generator and translators to a range of representations suitable for use with some theorem provers and model-checking tools. The state of the tool development project is reported on our WWW pages [TDS].

This document is concerned with the IOA simulator in particular. We describe the design of the simulator, the major issues regarding its implementation and also provide a set of examples to demonstrate how to use the simulator. The IOA simulator has been developed over a period of four years by a number of people contributing to its design and implementation [TDS]. It has been the subject of the MEng theses of the authors Anna Chefter [Che98], Antonio Ramirez [RR00] and Laura Dean [Dea01]. This document is intended to be a stand-alone reference for the IOA simulator and refers to the current implementation of the tools unless explicitly stated otherwise.

The idea behind the simulation of a single automaton is rather conventional. The IOA simulator runs selected executions of an I/O automaton on a single machine, generates logs of execution traces and displays information upon the user's request. The IOA Language allows users to express invariants for an automaton. The simulator checks whether these invariants proposed by users are true in the selected executions. The IOA simulator also has the capability to simulate pairs of I/O automata, allowing users to reason about the behavioral correspondence between automata at different levels of abstraction. The need for this style of reasoning typically arises when a system is designed by moving through the highest level to the lowest level in the abstraction hierarchy. In this case, users define a simulation relation which relates the two automata at two different levels and the IOA simulator checks whether this relation holds in the selected executions. The capability to perform paired simulation in this sense is a very useful feature in distributed system design and analysis.

1.2 Purpose of simulation

Formal correctness proofs for distributed systems can be long, hard or tedious to construct. Simulation can be used as a way of testing automata before delving into correctness proofs. The execution of an IOA automaton either reveals bugs or increases the confidence that an automaton works as expected.

The simulator can also assist users in constructing correctness proofs. By describing a system or an algorithm as an IOA program and simulating it, a user gains a better understanding of how it works. This can guide the strategy to be followed in proving correctness. Moreover, the invariants which are observed to be true for the simulated executions constitute candidates for useful lemmas in a full correctness proof.

The current implementation of the IOA simulator does not aim at providing quantitative information of the kind that would be useful for evaluating the performance of an algorithm under various conditions. However, it is conceivable that the IOA simulator be used for this purpose by means of some extensions to its design and implementation.

Simulation in general is an efficacious method for exposing possible deficiencies in the design of systems and algorithms which can lead to the correction of discovered errors, revision of proofs or tuning for better performance.

1.3 Design goals

A key challenge in the design of the IOA language has been to provide support for both simulation and verification in a unified framework. Nondeterminism is favorable in IOA because it 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. On the other hand, nondeterminism complicates simulation, which must choose particular executions. The design of a satisfactory mechanism for resolving nondeterminism is an essential issue concerning the design of the simulator. The approach adopted by the IOA simulator is described in greater detail in the following sections. We note here the properties that have been identified as desirable properties for the nondeterminism resolution mechanism:

- *Broadness.* It should provide several ways to resolve nondeterminism, each suited to different situations and applications. For instance, it should allow choices and transitions to be resolved as deterministic functions of the automaton's state, or using a pseudorandom number generator, or by querying the user, or any combination of these.
- *Extensibility.* It should be sufficiently open-ended that future developers and advanced users can tailor it to specific needs without too much effort. For instance, if a new datatype implementation is added to the simulator, it should be possible to add useful nondeterminism resolution mechanisms to go with it.
- *Usability.* It should be reasonably easy to use, and it should not place cumbersome demands upon the user. The resolution of nondeterminism is an absolute necessity for nontrivial uses of the simulator, and it would be unfortunate that a lack of attention to usability considerations should discourage its use.

1.4 How to use this document

The intended audience for this document is both users and developers of the IOA toolkit. The material has been organized so that it should be sufficient to read the first 5 sections to be able to use the IOA simulator and to understand the fundamental ideas behind its design. Section 6 is for readers who are familiar with the core IOA language and are interested in a formal presentation of the syntactic extensions made to support simulation. Section 7 is intended for tool developers; it gives an overview of the IOA simulator implementation.

2 I/O automata and the IOA language

This section includes a brief introduction to the I/O automaton model and the IOA Language. See [Lyn96, GLV01] for an in-depth introduction. We focus only on those notions and language constructs that are crucial for understanding the material in this document.

2.1 Theoretical background

An I/O automaton is a simple type of state machine in which the transitions are associated with named *actions*. The actions are classified as either *input,output*, or *internal*. The inputs and outputs are used for communication with the automaton's environment, whereas internal actions are visible only to automaton itself. The input actions are assumed not be under the automaton's control, whereas the automaton itself controls which output and internal actions should be performed.

An I/O automaton A consists of five components:

- a *signature*, which lists the disjoint sets of input, output, and internal actions of A;
- a (not necessarily finite) set of *states*, usually described by a collection of state variables;
- a set of *start* states, which is a non-empty subset of the set of all states;
- a *state-transition relation*, which contains triples (known as *steps* or *transitions*) of the form (state, action,state); and
- an optional set of *tasks*, which partition the internal and output actions of A.

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. That is to say automata are not able to block input actions from occurring. The *external* actions of an automaton consist of its input and output actions.

2.1.1 Executions and traces

An execution fragment of an I/O automaton is either a finite sequence $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_n, s_n$, or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$, of alternating states s_i and actions π_i such that s_i, π_{i+1}, s_{i+1} is a transition of the automaton for every $0 \leq i$. An execution is an execution fragment that begins with a start state. A state is *reachable* if it occurs in some execution. The *trace* of an execution is the sequence of external actions in that execution.

2.1.2 Properties and proof methods

Invariant assertions An *invariant* property of an automaton is any property that is true in all reachable states of the automaton. Invariants are typically proved by induction on the number of steps in an execution leading to the state in question.

Simulation proofs The I/O automaton model aims at providing support for system descriptions at multiple levels of abstraction. The process of moving through the series of abstractions, from highest level to the lowest level is called *successive refinement.* The top level may be a problem specification written in the form of an automaton. The next level describes the system in more detail with respect to the top level. However, the actions typically have large granularity, and simple data structures are used. Lower levels in the abstraction hierarchy correspond more directly to the most optimized implementation of the system. To prove that one automaton implements another one higher in the hierarchy, 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* facilitates this style of reasoning.

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 so that $f(a, b)$ holds.
2. If a is a reachable state of A, b is a reachable state of B, $f(a, b)$ holds and $a \xrightarrow{\pi} a'$, then there exists a state b' of B and an execution fragment β of B so 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.

Remark on terminology There is an unfortunate clash of terminology, due to the dual use of the term "simulation". Depending on the context, this term can refer either to the action of a simulator or to simulation relations as in Definition 2.1.

2.1.3 Composition

The composition operation allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving action π, so do all component automata that have π in their signatures.

A countable collection $\{S_i\}$ of signatures is said to be *compatible* if for all $i, j \in I, i \neq j$ all of the following hold:

- $int(S_i) \cap acts(S_j) = \emptyset$ where $int(S_i))$ denote the set of internal actions in S_i, and $acts(S_j)$ denotes the set of actions in S_j.
- $out(S_i) \cap out(S_j) = \emptyset$ where $out(S_i)$ and $out(S_j$ denote the set of output actions in (S_i) and (S_j) respectively.
- No action is contained in infinitely many sets $acts(S_i)$.

We say that a collection of automata is *compatible* if their signatures are compatible. The *composition* $S = \prod_{i \in I} S_i$ of a countable compatible collection of signatures $\{S_i\}$ is defined to be the signature with

- $out(S) = \cup_{i \in I} out(S_i)$
- $int(S) = \cup_{i \in I} int(S_i)$

- $in(S) = \cup_{i \in I} in(S_i) \setminus \cup_{i \in I} out(S_i)$

Now, the *composition* $A = \prod_{i \in I} A_i$ of a countable, compatible collection of I/O automata $\{A_i\}_{i \in I}$ can be defined as follows:

- $sig(A) = \prod_{i \in I} sig(A_i)$
- $states(A) = \prod_{i \in I} states(A_i)$
- $start(A) = \prod_{i \in I} start(A_i)$
- $trans(A)$ is the set of triples (s, π, s') such that, for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s_i{}') \in trans(A_i)$; otherwise $s_i = s_i{}'$
- $tasks(A) = \prod_{i \in I} tasks(A_i)$

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. If **pre** is not specified, then it is assumed to always hold. State changes are specified in terms of the initial state, the transition parameters, and optional additional parameters, which are chosen nondeterministically. 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.

The IOA language supports descriptions of systems composed from several interacting components based on the notion of composition in the theory of I/O automata.

The sample programs in this paper do not exploit the full generality of the language. We assume that the automata are pre-composed, and restrict ourselves to a subset of the language that consists of imperative features and nondeterministic choice statements constrained by **where** predicates.

2.3 Future research ideas

The current IOA language allows description of distributed systems without any timing-dependence. We are interested in extending the language with constructs to express timing behavior, including upper and lower bounds on times for various events, and program constructs such as timeouts. Various IOA tools, in particular, the simulator must also be extended to handle these new constructs. In the longer run we also aim to provide language support for describing and analyzing systems with probabilistic automata and hybrid automata.

3 Simulation of I/O automata

This section describes how the simulator is designed focusing on the IOA language support that it requires, and the algorithm that it follows to simulate an automaton. We do not treat details such as the management of operator and sort implementations. The reader is referred to Section 7 for further information about this and other software-related issues of the simulator.

3.1 Simulation and nondeterminism

IOA programs allow two kinds of nondeterminism: *implicit nondeterminism* which involves the scheduling of actions, and, *explicit nondeterminism*, which arises from **choose** statements, **choose** parameters and **choose** expressions in initial assignments. For example:

- an automaton can have multiple enabled actions in a given state;
- a given enabled action can have multiple transition definitions associated with it;
- a given transition definition can take arbitrary actual parameter values, as long as they satisfy its **where** clause; and
- a transition definition can contain one or more **choose** statements, each of which may evaluate to an arbitrary value that satisfies the constraint in the **where** clause.

3.2 Resolution of nondeterminism

From the point of view of an IOA automaton specification, the sources of nondeterminism can be regarded as a black box that can yield transitions to be scheduled and values to be assigned to statements which involve nondeterministic choice. Thus, the problem of resolving nondeterminism can be regarded as that of providing an algorithmic means of obtaining these values and transitions as the need for them arises during the simulation of an automaton.

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. Each such program is capable of providing values that resolve a choice, or determining the transitions to be scheduled, depending on the context. There is an NDR program corresponding to every **choose** statement in an automaton, and an NDR program for scheduling the actions of the automaton. We illustrate the key points of our approach by a series of examples based upon an automaton – `Chooser` – described as an IOA program.

Example 3.1. The automaton `Chooser` has two actions (`action1` and `action2`), and two state variables `chosen` and `did_choose` which is initially set to `false` to indicate that no integer has yet been chosen by the automaton. The transition definitions show that `action1` is always enabled. Its effect is to nondeterministically choose an integer greater than or equal to 10 and assign the variable `chosen` to this integer. It also sets the state variable `did_choose` to `true`. The semantics of the IOA language requires that the assignments to `chosen` and `did_choose` occur atomically. The transition definition for `action2` has a parameter, and the action is enabled when an integer has already been chosen and n is equal to that integer. The occurrence of `action2` has no effect on the state.

```
automaton Chooser
  signature
    output action1
    output action2(n: Int)
  states
    chosen: Int,
    did_choose: Bool := false
  transitions
    output action1
      eff chosen := choose x: Int where 10 ≤ x;
          did_choose := true
    output action2(n)
      pre did_choose ∧ n = chosen
```

This automaton exhibits both explicit and implicit nondeterminism. The **choose** statement in the definition of transition for `action1` is the source of explicit nondeterminism. After `action1` has occurred at least once, both `action1` and `action2(n)` become enabled where the actual parameter n is equal to the value chosen by `action1`. The possibility of more than one action being enabled is the source of implicit nondeterminism in this automaton.

3.2.1 NDR programs

To aid the simulator in resolving nondeterminism a user is required to augment the automaton specification with a **schedule** block and **det** blocks each of which embodies an NDR program. A program in a **schedule** or a **det** block is used respectively for resolving automaton transitions and for resolving the values of a **choose** statement. Note that this requires modification of the IOA language syntax as discussed in Section 6.

Example 3.2. The automaton `Chooser` can be augmented as below with NDR programs.

```
automaton Chooser
  signature
    output action1
    output action2(n: Int)
  states
    chosen: Int,
    did_choose: Bool := false
  transitions
    output action1
      eff chosen := choose x: Int where 10 ≤ x
                det do
                    % NDR program to be specified
                od;
          did_choose := true
    output action2(n)
      pre did_choose ∧ n = chosen
schedule do
    % NDR program to be specified
od
```

The NDR programs in **schedule** and **det** blocks can evaluate arbitrary IOA terms to decide which transitions to schedule, or which values to yield for a choice. Additionally, they can evaluate operators whose implementations perform pseudorandom number generation, or user prompting, to produce a result. Two forms of statements – **fire** statements and **yield** statements – have been introduced to IOA as essential building blocks of NDR programs.

3.2.2 Fire statements

Schedule blocks use **fire** statements to specify how the actions will be scheduled by the simulator. A **fire** statement specifies the parameters of an action and whether it is an input, output or an internal action. The parameters in these statements may depend on the values of state variables of the automaton. The NDR mechanism also supports **fire** statements with no arguments. These are useful under circumstances when it would be tedious to write a complete schedule by hand. When the simulator encounters a **fire** statement without arguments in an NDR context, it chooses an appropriate transition to schedule according to the following mechanism. It first examines in turn each locally-controlled transition definition of the automaton with no parameters. For each of them, it evaluates the precondition to see if it is enabled. It chooses one of the enabled transitions randomly and executes it.

In the special case of an automaton where all transitions are non-parameterized, the simulator can be run without a schedule block. At each step the simulator executes one of the enabled transitions. However, there are no guarantees about randomness or completeness. Note that we recommend the use of **schedule** blocks as part of a good programming discipline for simulation.

3.2.3 Yield statements

A **yield** statement is used to specify the values of choice in a **choose** statement. When the simulator encounters a **choose** statement, it starts executing the NDR program until it encounters a **yield** statement. At this point, it uses the value provided by the statement as the value of the **choose** statement. The current statement of the NDR program is recorded by the simulator so that the next time it encounters the same **choose** statement, the simulator does *not* start its NDR program from the beginning; rather, it resumes executing it where it left off. [2]

Example 3.3. This example illustrates the use of **yield** and **fire** statements in NDR programs. The particular **det** block we have added causes the choice to be resolved successively to 11, 12, and 13. The **schedule** block has been coded such that the simulator interleaves the executions of `action1` and `action2`.

```
automaton Chooser
  signature
    output action1
    output action2(n: Int)
  states
    chosen: Int,
    did_choose: Bool := false
  transitions
    output action1
      eff chosen := choose x: Int where 10 ≤ x
                det do
                      yield 10; yield 11; yield 12
                od;
          did_choose := true
    output action2(n)
      pre did_choose ∧ n = chosen
schedule do
    while true do
      fire output action1;
      fire output action2(chosen)
    od
od
```

It may appear surprising to have a nonterminating **while** loop in the schedule block. This, however, does not cause a problem since the simulator has been designed so that the number of simulation steps are specified by the user at the beginning of simulation. Section A.1 on page 43 shows the excerpts from the output of the simulator on the automaton `Chooser`. The simulator takes as command line arguments the number of transitions to simulate, the name of the automaton to simulate, and the name of a file containing the IOA specification of the automaton. For every step taken by the automaton (including the initialization step), the simulator reports the transition that was executed, and the state variables that changed. The sample output has been obtained by

[2] The semantics of yield and fire statements were inspired by the iterator construct in the programming language CLU [LAB+81].

simulating the automaton for 100 steps. The example in Section 5 gives a detailed explanation of how to use the simulator.

3.2.4 Labeling transition definitions

The IOA Language allows multiple transition definitions to share the same action type, name and actual parameter sorts. In the absence of a mechanism to disambiguate these definitions, specifying action names in **fire** statements alone would not be sufficient to resolve nondeterminism. As a solution to this problem, the simulator incorporates a facility whereby a user can augment action names with **case** indicators.

Example 3.4. The **case** indicator of the transition is local to the primitive automaton in which it is defined, and it can be a number or an alphanumeric identifier as shown in the example below.

```
automaton Undecided
  signature
    output hello
  states
    b: Bool
  transitions
    output hello case 1
      eff b := true
    output hello case 2
      eff b := false
schedule do
       while true do
            fire output hello case 1;
            fire output hello case 2
       od
od
```

3.2.5 Alternative methods of resolving nondeterminism

It is sometimes desirable to resolve choices and schedule transitions using pseudorandomness or user input as information. This issue can be addressed by providing extra operators that evaluate as random number generators and user prompters. One way to do this is to use a trait such as the one in Section B on page 51. Each of these operators is either currently implemented by the simulator, or is easy to implement with the current software support.

Example 3.5. This version of the Chooser automaton uses an operator that yields an integer between 20 and 30 rather than specifying the integers as was the case in Example 3.3.

```
uses NonDet
automaton Chooser
  signature
    output action1
    output action2(n: Int)
  states
    chosen: Int,
    did_choose: Bool := false
  transitions
    output action1
      eff chosen := choose x: Int where 10 ≤ x
                det do
                      yield randomInt(20,30)
```

```
              od;
          did_choose := true
    output action2(n)
      pre did_choose ∧ n = chosen
schedule do
    while true do
      fire output action1;
      fire output action2(chosen)
    od
od
```

Note that it is also possible to prompt the user to choose an integer at the point where the operator `randomInt` is used in this example.

3.2.6 Simulation errors

The simulator requires NDR programs to only **fire** transitions that are enabled, and **yield** choice values that make the corresponding **where** clause true. If the simulator encounters a situation where either of these conditions does not hold, it issues an error message and halts the simulation.

3.3 The simulator algorithm

So far, we have pointed out that it is necessary to resolve nondeterminism to be able to simulate IOA programs. There are, however, other requirements for an IOA program to be in the right form for simulation. The users are expected to transform programs into this required restricted form before using the IOA simulator.

3.3.1 Simulability conditions for programs

Quantifiers The simulator has the ability to handle quantifiers only when the quantified variable is of enumeration type. This implies that the variable has a finite number of possible values. Existential or universal quantifiers which do not satisfy this condition are not permitted anywhere in the IOA automaton to be simulated. The effect of an existential quantifier can often be achieved using a suitably constrained **choose** statement as described in [Che98], thereby reducing the problem of evaluating such quantifiers to the problem of nondeterminism resolution for **choose** statements. Evaluating universal quantifiers would require an essentially different mechanism.

Transition parameters There are restrictions on the actual parameters in transition definitions: each of them must be either a pure variable, or a term that contains no variables, so that it evaluates to a constant. As explained in [Che98], this is not a drastic restriction, since expression parameters can be replaced by variables that are suitably constrained by the **where** clause of the transition. It would not be difficult to modify the current implementation to remove this constraint, but some corresponding changes to the NDR mechanisms would be necessary.

Looping constructs No **for** loops are permitted anywhere in the automaton to be simulated. It is often possible to use a **while** loop instead. For example, **for** `i:Nat` **where** `i < 20` **do** ... **od** can be replaced by **while** `i < 20` **do** `i:= i+1; ...` **od**. Note that **while** does not incorporate a mechanism for declaring a variable; the variable `i` must be declared and initialized outside the loop.

Composition The simulator only supports primitive automaton specifications. There is a project in progress on the development of a tool which takes an IOA automaton composition specification as an input, and transforms it to an equivalent IOA specification of a primitive automaton. Once this composer implementation is complete it can be used in conjunction with the simulator. Composite automata can be simulated by providing the necessary NDR programs for the output of the composer.

Data types The simulator currently has implementations for several built-in primitive IOA types (`Bool, Natural, Real, Char, String`) and it supports user-defined types formed from the constructors `Array` (for one-dimensional arrays), `Seq` (sequence), `Set, Mset`(multiset), and `Map` constructors and syntactic shorthands **enumeration, tuple**, and **union** shorthands, and those formed from the. These types, constructors and shorthands are described in the IOA Manual [GLV01]. There is currently no implementation for the two dimensional use of `Array`. Specifications and implementations for the parameterized datatypes `Stack, Tree` and `PQ`(priority queue) are also available for use with the simulator even though they are not yet a part of the language specified in the IOA Manual [GLV01]. Note also that it is possible to add new data types to the Simulator as explained in Section 7.

3.3.2 Pseudocode

A good way to understand how the simulator interprets NDR programs is through a description of the algorithm that it follows. On Page 11 we present a table which summarizes the abbreviations and the notation we use in describing the algorithm. Page 12 includes the pseudocode description of the simulator algorithm which is organized into three procedures. The main one is Simulate(A), where A is the primitive automaton specification to be simulated. This procedure in turn uses two auxiliary ones, ExecuteSched and EvalChoice also presented in the figure. The algorithm does not describe the details of evaluating IOA programs or terms but focuses on the NDR mechanisms. Evaluating a term requires every operator in the term to have a simulator implementation; refer to Section 7 for the details on matching operators and sorts with their implementations.

Notation

$A.ndr$	The schedule NDR program for automaton specification A.
$A.pc$	A program counter for $A.ndr$. Its value can be a statement in $A.ndr$ or $null$.
$A.invs$	The list of invariants of A.
$A.simpleTrans$	The set of transition definitions in A with constant actual parameters.
$t.pre$	The **pre**condition term for a transition definition t.
$t.where$	The **where** term for a transition definition t.
$t.eff$	The **eff**ect program for a transition definition t.
$c.ndr$	The choice NDR program for a **choose** statement c.
$c.pc$	A program counter for $c.ndr$. Its value can be a statement in $c.ndr$ or $null$.
$c.var$	The dummy variable in a **choose** statement c.
$c.where$	The where term in a **choose** statement c.
$trans(A,t,n,c)$	The transition definition of type t, name n and case label c in automaton A.
$eval(t)$	The result of evaluating a term t.

```
Simulate($A$) [$A$: IOA primitive automaton]
  initialize a program counter $c.pc$ for each **choose** statement $c$ in $A$
  initialize a program counter $A.pc$ for the **schedule** block of $A$
    while $A.pc \neq null$ do
      call ExecuteSched($A$,$A.pc$)
      advance $A.pc$ to the next statement in $A.ndr$

ExecuteSched($A, s$) [$A$: IOA primitive automaton, $s$: statement in $A.ndr$]
  if $s$ is not a **fire** statement then execute $s$
        ($s$ is an assignment, a conditional, or a **while** construct;
        the semantics for these types of statements are the obvious ones)
  else if $s =$ ``**fire** $actionType$ $actionName$($actionActuals$) **case** $c$'' then
    let $t := trans(A, actionType, actionName, c)$
    assign $actionActuals$ to the formal parameter variables of $t$
    if $eval(t.pre) = true$ and $eval(t.where) = true$ then
      execute the statements in $t.eff$ following IOA semantics;
      when a **choose** statement $c$ needs to be evaluated, call EvalChoice($c$)
    else halt with an error
    for each $t \in A.invs$ such that $eval(t) = false$ do
      issue an invariant failure warning
  else if $s =$ ``**fire**'' then
    let $S = \{t \in A.simpleTrans | \ eval(t.pre) = true\}$
    if $S \neq \emptyset$ then
      choose $t \in S$ uniformly at random
      execute the statements in $t.eff$ following IOA semantics;
      when a **choose** statement $c$ needs to be evaluated, call EvalChoice($c$)

EvalChoice($c$) [$c$: **choice** statement]
  forever do
    if $c.pc$ is not a **yield** statement then
      execute $c.pc$ ($c.pc$ is an assignment, a conditional, or a **while** construct)
      advance $c.pc$ to the next statement in $c.ndr$
    else if $c.pc$ is of the form ``**yield** $t$'', where $t$ is a term then
      let $v = eval(t)$
      assign $v$ to $c.var$
      if $eval(c.where) \neq false$ then
        advance $c.pc$ to the next statement in $c.ndr$
        exit EvalChoice
      else halt with an error
```

Figure 1: Simulator Algorithm

3.4 Invariant checking

The simulator has the capability of checking whether the invariants of an automaton, stated using the IOA syntax, hold throughout an execution. This is done simply by evaluating each of the invariants found in the IOA specification after each transition is executed, and issuing a warning message if any of them fail. The ExecuteSched routine of the pseudocode of the algorithm presented in Section 3.3 includes a part for dealing with invariant checking.

Example 3.6. The code in this example is an IOA specification of an automaton, along with two proposed invariants of its state and suitable NDR programs.

```
automaton Fibonacci
  signature
    internal compute
  states
    a: Int := 1,
    b: Int := 0,
    c: Int := 1
  transitions
    internal compute
      eff
        a := b;
        b := c;
        c := a + b
invariant of Fibonacci:   % true invariant
  a + b = c
invariant of Fibonacci:   % false invariant
  a - b = c
```

Section A.2 on page 44 gives the simulator output for 5 steps of execution. It shows that one of the invariants did not hold for this particular execution.

3.5 Dynamic detection of invariants

This section describes the connection between the IOA simulator and Daikon – an invariant discovery tool developed by the Program Analysis Group at the MIT Laboratory for Computer Science [PAG].

3.5.1 Daikon

Daikon is a dynamic program analysis tool which extracts information from executions of a program. As input, Daikon requires a set of declarations and data traces. A declaration file contains lists of program points considered interesting to users and a list of variables in scope at each program point. Data trace files record information about the values the variables take on during execution. For each execution of a program point, the trace file contains the name of the point and the values of the variables at that point. The output generated by Daikon is a list of invariants detected to hold in all recorded executions. These are only potential invariants in that Daikon cannot guarantee their truth for all possible executions.

3.5.2 Purpose of connecting IOA to Daikon

There are mainly two motivations for connecting the IOA simulator with an invariant discovery tool such as Daikon. First of these concerns correctness proofs for automata. If the discovered invariants turn out to be verifiable, they can assist the proofs in several ways. One possibility is that Daikon discovers invariants that are not readily detectable by users. In this case Daikon helps proofs by discovering those invariants that would have remained unnoticed by users. At the other extreme lie the invariants that are easily detectable by users even without the help of Daikon. The automatic discovery of such invariants is considered also useful, since it saves users the effort of finding and formulating these simple invariants.

Second, Daikon might suggest invariants which are known to be not always true, pointing to shortcomings in the simulation. The IOA code and NDR programs should then be examined to correct errors or to increase the simulator's coverage of possible executions.

3.5.3 Interface to Daikon

Daikon has initially been designed to discover invariants for sequential programs written in languages such as C or Java. It is however possible to make use of Daikon in discovering invariants for programs written in other languages so long as it is supplied with suitable declarations and data traces regarding a program. The simulator provides the necessary machinery for this. In the preceding sections we have described how the IOA simulator executes I/O automata written in the IOA Language. The necessary input for Daikon can be generated by the simulator by recording data traces while executing I/O automata. This is achieved by running the simulator with a special option (–daikon) as described in Section 5.

When run with the above mentioned option, the IOA simulator generates a declaration file which declares a program point for the entry and exit of every transition and a program point for the automaton. Declaring entry and exit of every transition point as an interesting program point allows Daikon to infer how a transition's pre-state relates to its post-state. The program point at the top level allows Daikon to detect invariants that hold at all times, not just at certain entry and exit points in the automaton. Technical issues regarding the implementation can be found in [Dea01, WS01].

3.6 Future research ideas

In this section we describe how we intend to continue our work on the IOA simulator. Our experiments convince us that the current state of the IOA simulator allows it to be used for nontrivial tasks in distributed system design and analysis. The future research will mostly concern user convenience and keeping the simulator in tandem with the extensions to the IOA language.

3.6.1 Scheduling policies

The users of the IOA simulator are required to encode scheduling policies explicitly by means of NDR programs. It would be possible alleviate this burden on the users if the simulator was given the capability to make scheduling decisions. We outline below a method for enhancing the IOA simulator with such a capability.

The syntax and the semantics of schedule blocks are redefined so that the users are required only to resolve explicit nondeterminism, provide a list of conditional clauses that specify the set of selected transitions and their parameter values. They select a scheduling policy prior to simulation and communicate this choice to the simulator. Whenever multiple transitions are enabled during the execution, the scheduler selects a transition to be executed according to the scheduling policy that has been chosen by the user.

This idea has appeared in Chefter's design of the simulator, however it is not supported by the IOA simulator yet. According to this design the user has a choice of three scheduling policies: randomized, round-robin, and one based on time estimates for each action. Moreover, the user is required to specify a weight (w) or time estimate for each transition to be used by the scheduler in the case of choosing the randomized policy or the policy based on time estimates respectively.

For the randomized scheduler, the simulator computes the total t of the weights of all specified transitions, and at each step of the execution selects a transition with weight w with probability w/t.

The round-robin scheduler keeps track of the number of times a transition was enabled but not selected for execution and maintains a queue of these counts. It always selects the transition with the greatest count. The count is reset to zero after the transition is executed.

In time based scheduling time estimates are used for determining the probability of each action being scheduled such that the smaller the time estimate, the higher the probability that the action will be scheduled. Time estimates allow one to model the running of a system on multiple processors with different speeds. For example, if an action is intended to be run on a fast processor the time estimate associated would be smaller than that of other actions which are intended to be run on slower processors. Similarly, time estimates can be used to model computation latency or the rate at which an environment generates actions. Specifically, if times for n actions are given by n integers $time_1, time_2, \ldots time_n$, then the scheduler determines which of the n actions to perform by the following procedure:

1. Find the least common multiple m of $time_1, time_2, \ldots time_n$
2. Assign a weight to each selected action as follows:

$$weight_i = (m/time_i)/\sum_{j=0}^{n-1}(m/time_j).$$

3. Divide the interval $[0 \ldots 1]$ into n parts

$$[0 \ldots weight_0], [weight_0 \ldots weight_0 + weight_1], \ldots, [\sum_{j=0}^{n-2}(weight_j) \ldots 1]$$

and schedule the ith action if the random number is in the range

$$[\sum_{j=0}^{i-1} \ldots \sum_{j=0}^{i} weight_j].$$

The I/O automaton task partition can be thought of as an abstract description of threads of control within an automaton, and is used to define fairness conditions such that each of the tasks is given fair turns during execution. The simulator does not support task partitions, however it would be useful to devise a two-level mechanism for scheduling where the first level selects the next task to be scheduled and the second level selects a particular action within a task.

3.6.2 NDR libraries

The current mechanism for nondeterminisim resolution might lead to repetitive code fragments scattered over the automaton description (one NDR program for each **choose** statements) and complex **schedule** blocks. More important, it is the user who has to provide these programs. If the IOA simulator provided a library of NDR programs or some default NDR programs, the users would be relieved from having to do this. For each commonly encountered sort in IOA programs, such as natural numbers or booleans, the simulator could specify a default NDR program to be used when no NDR program is provided by the user. The similar idea applies to the predicates in **choose** statements. For example, many **choose** statements have **where** predicates that restrict the range of the chosen value to some fixed finite set of numbers. It would be possible to determine some patterns for predicates such as $p : \texttt{Int} \leq q \wedge q : \texttt{Int} \wedge r : \texttt{Int}$ and have the simulator provide a library of NDR programs which resolve nondeterminism such that the predicate holds.

3.6.3 Alternatives to NDR programs

It is possible to resolve some of the nondeterminism in an automaton to be simulated by modifying its IOA specification. For example, the user can augment the automaton with new state variables containing scheduling information, can add extra constraints involving the new scheduling variables to the preconditions of transitions, and can add extra statements to the effects of transitions to maintain the scheduling variables. This conversion must be done manually, without the help of the NDR programs. We are considering the relative advantages of resolving nondeterminism with NDR programs as explained throughout this document or within the IOA language itself as mentioned above. We are planning to continue our work by evaluating the effects of alternative nondeterminism resolution schemes on the IOA programs with respect to user convenience, reusability of code within the toolkit and elegance.

3.6.4 Theorem proving using Daikon-detected invariants

A group of us are investigating how to make invariants discovered by Daikon more relevant to proofs of correctness of distributed systems. Toh Ne Win has recently finished an experiment on using Daikon-discovered invariants in the verification of a mutual exclusion algorithm [Win02]. By carrying out similar but more advanced experiments, we aim to identify when an invariant should be considered useful. Our ultimate aim is to make correctness proofs more automatic by feeding these invariants into the theorem prover. Our current efforts are based on the Larch Prover. However, we are potentially interested in using other theorem provers such as ACL2, Isaballe or HOL.

4 Paired simulation

In the study of distributed systems, it is common for complex systems to be analyzed through successive refinements: in the presence of an abstract specification A, one would like to show that another specification B is an *implementation* of A. If A and B are I/O automata, this is modeled by the statement that $traces(B) \subseteq traces(A)$.

To prove a statement of this form, it is almost inevitable to use an argument by induction on the length of a finite prefix of an execution of B. This inductive reasoning on automaton executions has been abstracted, yielding the method of simulation relations. Using this method, one seeks to construct a simulation relation f from B to A. For a formal definition of simulation relations see Section 2.

4.1 Simulation relations

The IOA Language includes syntax for asserting simulation relations between automaton specifications. One of the goals of IOA is to provide software tools to assist the analysis of I/O automata. For example, given a proposed simulation relation f from B to A, it would be useful to test its validity when restricted to a particular execution of B. As in the case of invariants, a single execution in which f is observed not to hold would suffice to show that f is invalid. While continued verification of f in different executions of B does not prove the correctness of f, it does provide empirical evidence that f may be true, before the user spending the necessary effort to prove its correctness.

In this section, we describe how the simulator described so far in the paper was extended to allow simulation of a pair of automata related by a mathematical simulation relation. The key problem here is the following: the simulation relation itself, being merely a predicate that relates the states of two automata, is not sufficient to specify 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, and even if there is only one, it can be difficult to find it. From this point of view, 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 non-determinism. Not surprisingly, the problem of programmatically specifying a step correspondence admits a similar solution.

4.2 Encoding step correspondences

A step correspondence needs to specify, for a given low level transition, a high level execution fragment such that the simulation relation holds between the respective final states of the transition and the execution fragment. 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 current syntax of the IOA construct **forward simulation** was extended to include a new section called **proof** for specifying the step correspondence. This section contains one entry for each possible transition definition in the low level automaton, and each entry encodes an algorithm for producing a high level execution fragment, using a program similar to the NDR programs used in automaton **schedule** blocks. In addition to these entries, the **proof** section also 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.

Figure 2 on Page 18 shows the general high level structure of a simulation proof encoded using this language. Note that this syntax extends the syntax for forward simulation relations in IOA. Some of the sections in the **proof** block have a more flexible syntax than is depicted here, and some can be omitted; refer to Section 6 for the detailed grammar. The **states** block introduces auxiliary variables used in the proof, and their initial values. The **initially** block specifies how to initialize the state variables of the specification automaton as a function of the implementation automaton's initial state, so as to satisfy the simulation relation.

Each $proofEntry_i$ is either the keyword **ignore** or a *proof program*, surrounded by **do** and **od** delimiters. Such a program is essentially an NDR program, of the form allowed in an automaton's **schedule** block, except that the **fire** statements must now provide additional information to resolve the **choose** statements of the specification automaton. If a proof program is present, the simulator will execute it from beginning to end to produce a high-level execution fragment for that case, using the **fire** statements to schedule transitions in the specification automaton. A proof entry equal to **ignore** is equivalent to a proof program with no statements, and it is used to represent an empty high-level execution fragment.

The **fire** statements allowed in proof programs have the structure depicted in Figure 3 on page 18. This general **fire** statement has the meaning: "schedule the transition of type *actionType*, name *actionName* with actual parameters *actionActuals*, using the values of the terms $term_1$ to $term_n$ to resolve the **choose** statements in the effect of the transition having dummy variables v_1 to v_n". If present, the *caseId* label is used to disambiguate between transition definitions with the same signature.

This design imposes a constraint not present in the single automaton case: it must be required that, for a given transition definition in the specification automaton, the **choice** statements in it have dummy variable names which are distinct. While in general it is undesirable to place unique-naming constraints for local dummy variables, we justify this design decision by arguing that, in

```
forward simulation
    from $autImpl$ to $autSpec$ :
    $simPredicate$
    proof
      states
        $auxVar_1 : sort_1$,
        $auxVar_2 : sort_2$,
        .
        .
        .
        $auxVar_m : sort_m$,
      initially
        $var_1 := term_1$;
        $var_2 := term_2$;
        .
        .
        .
        $var_n := term_n$
      for $actType_1$ $actName_1(actFormals_1)$
              case $caseId_1$
               $proofEntry_1$
    for $actType_2$ $actName_2(actFormals_2)$
              case $caseId_2$
               $proofEntry_2$
              .
              .
              .
      for $actType_p$ $actName_p(actFormals_p)$
              case $caseId_p$
               $proofEntry_p$
```

Figure 2: Syntax of step correspondence

```
fire $actionType$ $actionName(actionActuals)$
    case caseId
    using $term_1$ for $v_1$,
          $term_2$ for $v_2$,
          .
          .
          .
          $term_k$ for $v_k$
```

Figure 3: **fire** statements in **proof** blocks

the case of paired simulation, these are not just dummy variables, but serve also as natural names for the choices in a high-level transition. An alternative design would be to add syntax for explicitly naming the **choose** statements.

Example 4.1. The automaton `GreeterSpec` is a specification for automata that produce the output action `hello` any, perhaps infinite, number of times. The automaton `FiniteGreeter` is a specialization of this – an automaton that only produces a finite (bounded by the value of `maxGreets`) number of `hello` outputs. Note the use of dummy variable `sg` in the **choose** statement. `FiniteGreeter` has exactly one choice point, which occurs in its initialization of the `maxGreets` variable. To be able to simulate it, it has been augmented with an NDR program that yields 100 as the value of choice.

```
axioms NonDet

automaton GreeterSpec
  signature
    output hello
  states
    stillGoing: Bool
  transitions
    output hello
      pre stillGoing
      eff stillGoing := choose sg

automaton FiniteGreeter
  signature
    output hello
  states
    maxGreets: Int choose x:Int det do yield 100 od,
    count: Int := 0
  transitions
    output hello
      pre count < maxGreets
      eff count := count + 1

forward simulation
 from FiniteGreeter to GreeterSpec :
   GreeterSpec.stillGoing ⇔
       (FiniteGreeter.count < FiniteGreeter.maxGreets)
 proof
   initially
     GreeterSpec.stillGoing :=
            (FiniteGreeter.count < FiniteGreeter.maxGreets)
   for output hello do
     fire output hello
      using (FiniteGreeter.count < FiniteGreeter.maxGreets) for sg
  od
```

The **forward simulation** block embodies a simulation predicate, which states that the value of the variable `stillGoing` for automaton `GreeterSpec` is required to be `true` if the value of `count` in automata `FiniteGreeter` has not reached the value of `maxGreets` yet, and `false` otherwise. The **proof** block initializes the value of `stillGoing` and states the step correspondence suggested by the user. According to the user, each `hello` action executed by the low-level automaton (`FiniteGreeter`), can be mimicked by a `hello` action of the high-level automaton `Greeter` if the dummy variable is chosen to be the value of the predicate (`FiniteGreeter.count < FiniteGreeter.maxGreets`). It is the simulator's responsibility to check whether the simulation predicate holds and the traces of the low-level and high-level executions are the same.

Section A.3 on page 45 contains the output of the paired simulator for 100 steps. As in the case of non-paired simulation, it outputs the transitions taken and state variables modified for every step of the implementation automaton. In addition, it outputs the transitions of the specification automaton induced by each implementation step. For each transition taken in either automaton, the simulator outputs the variables that were changed by the transition's effect. The absence of simulator error messages in the output indicates that the simulation relation was verified to hold, in this particular run, with this proposed step correspondence. We refer the reader to Section 5 for a detailed description of how to run the paired simulator.

4.3 The paired simulator algorithm

In this section we present the pseudocode for the paired simulator on pages 21 and 22, as we did in Section 3.3 for the single automaton case. The pseudocode is organized into several procedures, of which SimulatePair is the main one. The reader is referred to Page 21 for the abbreviations and the notation used.

The procedure SimulatePair invokes the algorithm for single-automaton execution described in Section 3.3, except that it calls procedure ExecCorresponding for every low-level transition t that is scheduled. The procedure ExecCorresponding follows the proof program associated with t in the **proof** block of the simulation relation, executing each of the high level transitions determined by **fire** statements. In addition, ExecCorresponding verifies that the induced high level transitions have the same trace as t, and calls CheckSimRel to determine if the simulation relation holds at the end of the step. The procedure ExecSpecEffect, called by ExecCorresponding for each high-level transition, executes the effect program of the transition as in the single-automaton case, except that procedure EvalSpecChoice is called for every explicit choice. The latter procedure evaluates a **choose** statement using the value provided in the **using** part of the **fire** statement that determined the high level transition, provided that it satisfies the **where** predicate.

Notice that the low level step is taken in full before its corresponding proof entry is examined, and the prior state of the low level automaton is not recorded. This means that the proof program can only refer to the low level state *after* the low level step has taken place. Nevertheless, it is easy to modify an implementation automaton to make it keep track of relevant parts of its old state, or of the choices it makes. In this way, the proof can refer to this information, and the language can be very expressive. A possibility for future expansion is to extend the syntax so that it can refer explicitly to the state before and after the low level step, and to the choices taken during the step.

4.4 Future research ideas

There are many directions for future work on the paired simulation tool. We present below some suggestions for possible projects.

4.4.1 Improving the step correspondence language

The language described in this section is already substantially flexible, and it might be argued that together with auxiliary automaton state variables and auxiliary variables in the step correspondence, it allows one to express most of what is usually expressed in simulation proofs. However, to make easier to use, it might be desirable to have explicit syntax for:

- referring to state variable values both before and after the low-level transition, and,
- referring to the actual value to which an explicit choice was resolved in the low-level automaton.

Notation

$R.proof$	The **proof** block in simulation relation R
$R.impl$	The implementation-level automaton in simulation relation R
$R.spec$	The specification-level automaton in R
$t.pre$	The **pre**condition term for a transition definition t.
$t.where$	The **where** term for a transition definition t.
$t.eff$	The **eff**ect program for a transition definition t.
$c.var$	The dummy variable in a **choose** statement c.
$c.where$	The where term in a **choose** statement c.
$trans(A, t, n, c)$	The transition definition of type t, name n and case label c in automaton A
$eval(t)$	The result of evaluating a term t.
$proofProg(R, t)$	The proof program corresponding to t in $R.proof$. t must be a transition of $R.impl$

```
SimulatePair($R$):
[$R$:  IOA simulation relation]
  let $A := R.impl$, $B := R.spec$, $p := R.proof$
  call Initialize($R$)
  simulate $A$ as described in Section 3, except that:
    for each transition $t$ executed in $A$
      call ExecCorresponding($R$,$t$)

Initialize($R$):
[$R$:IOA simulation relation]
  let $A := R.impl$, $B := R.spec$, $p := R.proof$
  initialize the state of $A$ (using its NDR mechanism if necessary)
  initialize the auxiliary variables in the states block of $p$
  initialize the state of $B$ according to the initially block of $p$
  call CheckSimRel($R$)

ExecCorresponding($R, t$):
[$R$:  IOA simulation relation,
 $t$:  a transition of $R.impl$]
  $p := proofProg(R, t)$
  let $\ell$ be an empty sequence of transitions
  for each statement $s$ in $p$ do
    if $s$ is not a fire statement then
      execute $s$ ($s$ is an assignment, a conditional, or a while construct)
    else
      $t' := trans(S.spec, actionType, actionName, caseId)$
      call ExecSpecEffect($R, s, t'$)
      append $t'$ to $\ell$
  call CheckSimRel($R$)
  if $trace(\ell) \neq trace(t)$ then
    halt with an error
```

Figure 4: Paired Simulator Algorithm (1)

```
ExecSpecEffect($R, s, t$):
[$R$:IOA simulation relation,
 $s$:a **fire** statement of the form given in Figure 3,
 $t$:the transition of $R.spec$ corresponding to $s$]
  assign $actionActuals$ to the formal parameters of $t$
  if $eval(t.pre) = true$ and $eval(t.where) = true$ then
    execute the statements in $t.eff$ following IOA semantics;
    when a **choose** statement $c$ needs to be evaluated, call EvalSpecChoice($R, s, t, c$)
  else
    halt with an error

EvalSpecChoice($R, s, t, c$)
[$R$:IOA simulation relation,
 $s$:a,**fire** statement of the form given in Figure 3,
 $t$:the transition of $R.spec$ corresponding to $s$,
 $c$:a **choose** statement in $t.eff$]
  let $r := eval(term_i)$, where $v_i$ is the name of $c.var$
  assign $r$ to $c.var$
  if $eval(c.where) = false$ then
    halt with an error

CheckSimRel($R$)
[$R$:IOA simulation relation]
    if $eval(R.pred) = false$ then
      halt with an error
```

Figure 5: Paired Simulator Algorithm (2)

Neither of these two additions should be hard to implement. For example, prior and posterior values of variables could be distinguished with a prime decoration on variable names. References to low-level explicit choice values could be done using another unique-naming-per-transition convention, this time in the low-level automaton.

4.4.2 Interfacing with a computer-assisted theorem prover

The paired simulator may provide counterexample executions where the proposed step correspondence does not hold, but it will never completely certify the proof, even if it provides empirical evidence of its correctness after multiple simulations. However, a version of this language could be used as an interface between the simulation relation stated in IOA and a theorem prover: the proof program can be used to drive the theorem prover in the major overall steps of the proof, reducing the amount of routine work that the user has to do. We refer the reader to [KCD$^+$] for an example that illustrates the promise of this direction.

4.4.3 Adding syntax for providing a complete proof

As it stands, the **proof** block is not a really a proof, since it is missing the reasoning that shows that each high-level execution fragment produced by a **for** block in the proof preserves the simulation relation, assuming the relation held true in the immediately preceding state. An interesting project would be to add syntax that would allow the inclusion of this reasoning, in a form suitable for automated proof verification.

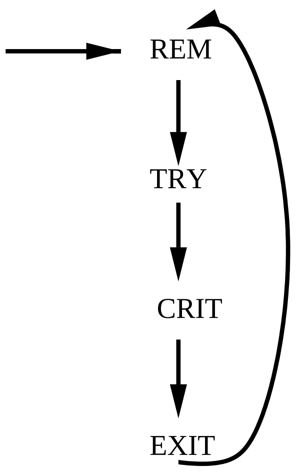


Figure 6: Cycle of regions for a single user

5 Mutual exclusion: A Tutorial example

In the preceding sections we introduced the basic concepts concerning the simulation of I/O automata and presented simple examples to illustrate the simulation language (an extension of IOA) supported by the IOA simulator. This section is intended to serve as a tutorial for using the IOA toolkit for simulating IOA programs. The instructions for obtaining the toolkit can be found at URL `http://theory.lcs.mit.edu/tds/ioa.html`.

We take a well-known problem in distributed algorithms research – the mutual exclusion problem – and proceed with the reader through multiple levels of abstraction in specifying the problem and deriving a low-level algorithm that implements mutual exclusion. We use the simulation tools to check that our algorithms work as expected and to increase our confidence in the correctness of the proposed simulation relations between different levels in the abstraction hierarchy.

5.1 The Mutual exclusion problem

The mutual exclusion problem involves the allocation of a single, indivisible, non-shareable resource among n processes. The resource could be, for example, 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*, and 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* (R) to its *trying region* (T), then to its *critical region* (C), then to its *exit region* (E), and then back to its remainder region. This cycle is shown in Figure 6.

In our example, we consider mutual exclusion algorithms within the shared memory model [Lyn96]. The shared memory system contains n processes, numbered $1, \ldots, n$. The inputs to process i are the try_i action which models a request for access to the resource by process i, and the $exit_i$ action, which models an announcement that process i is done with the resource. The outputs of process i

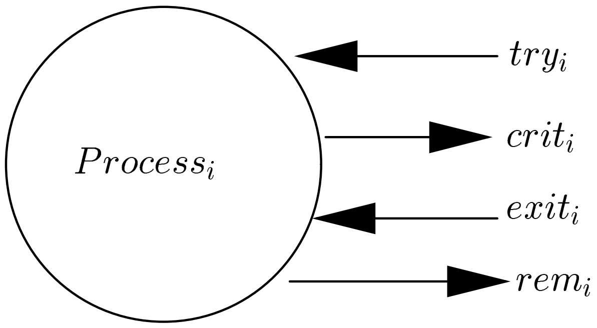


Figure 7: External interface of a process

are $crit_i$ which models the granting of access to process i, and rem_i which tells the process i that it can continue with the remainder of its work.

The $try, crit, exit$, and rem actions are the only external actions of the shared memory system. The processes are responsible for performing the trying and exit protocols. The external interface of process i is depicted in Figure 7.

5.2 Specification of mutual exclusion for three processes

The automaton `Mutex` below is the IOA specification for a mutual exclusion service in a system of three processes.

```
type Index = enumeration of p1, p2, p3

type Region = enumeration of rem, try, crit, exit

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

Explanation of code The code above assumes that the processes in the system are referred to by indices `p1`, `p2` and `p3` and the regions which constitute the cycle used in modeling the execution of a process are called `rem`, `try`, `crit` and `exit`. The definitions for types `Index` and `Region` are used to express these assumptions in IOA.

The signature of `Mutex` corresponds to the expression of the the external interface in the IOA language of a process shown in Figure 7. The state variable `regionMap` maps process indices to regions and is used to keep track of the current region of a process. Each process is assumed to be in its remainder region initially, hence the initialization of `regionMap` to `constant(rem)`.

The transition definitions are mostly self-explanatory. Each action causes the variable `regionMap` to be updated to record the region that is entered upon its execution. The transition definition for `crit` warrants more attention as it is this definition which 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 which is also in region `crit`.

5.2.1 The Environment

We have hitherto assumed that each process obeys the cyclic region protocol. 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 $try_i, crit_i, exit_i, rem_i, try_i, \ldots$ In this section we no longer assume but enforce the condition that the interaction of the automaton `Mutex` with its environment is well-formed by specifying the behavior of the environment by means of the automaton `Env`. The signature of `Env` is similar to that of `Mutex`. The point to notice is that the input actions of `Mutex` are output actions for `Env` and the output actions of `Env` are input actions for `Mutex`.

```
type Region = enumeration of rem, try, crit, exit

type Index = enumeration of p1, p2, p3

automaton Env
  signature
    output try(p: Index)
    input crit(p: Index)
    output exit(p: Index)
    input rem(p: Index)
  states
    regionMap: Array[Index, Region] := constant(rem)
  transitions
    output try(p)
      pre regionMap[p] = rem
      eff regionMap[p] := try
    input crit(p)
      eff regionMap[p] := crit
    output exit(p)
      pre regionMap[p] = crit
      eff regionMap[p] := exit
    input rem(p)
      eff regionMap[p] := rem
```

5.2.2 Well-formed interaction with the environment

The automaton `MutexEnv` below is an automaton which has been obtained by composing `Mutex` and `Env` according to the definition of composition from Section 2. The resulting automaton `MutexEnv` is the IOA specification of mutual exclusion for three processes where the well-formedness of interaction with the environment is guaranteed. The invariant at the very end asserts mutual exclusion.

```
type Index = enumeration of p1, p2, p3

type Region = enumeration of rem, try, crit, exit
```

```
automaton MutexEnv
  signature
    output try(p: Index)
    output crit(p: Index)
    output exit(p: Index)
    output 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

  invariant of MutexEnv:    % asserts mutual exclusion
    ∀ p: Index
      (regionMap[p] = crit
        ⇒ ∀ u: Index (p ≠ u ⇒ regionMap[u] ≠ crit))
```

5.2.3 Syntax and semantic checking with `ioaCheck`

Each IOA program needs to pass through a syntax checking phase before it is subjected to further study with back-end tools such as the simulator. The tool for syntax checking can be used by running the shell script `ioaCheck`. Note that this program also performs some semantic checks on the code. To check your code with `ioaCheck`:

1. Place the code in a file with extension .ioa. For example: MutexEnv.ioa
2. At the command line type
   $>$ `ioaCheck MutexEnv.ioa`

The result of using `ioaCheck` without any options is either a message on the standard output that indicates a successful check (`Finished checking specifications`) or errors. The command `ioaCheck` can also be used to check LSL specifications placed in a file with the extension.lsl. The following is the list of options available for running `ioaCheck`.

```
Usage
   ioaCheck [option] source-file
Options
  -il             translate to intermediate language
  -p              prettyprint source files
  -path <dirlist> use <dirlist> to find source files (default '.')
  -sorts          print sorts in first source file (LSL only)
  -syms           print symbols in first source file (LSL  only)
  -debug          print debugging information
  -verbose        print verbose debugging information
```

5.3 Levels of abstraction and simulation

In this section we present the IOA code of two algorithms that implement mutual exclusion specified by the automaton `MutexEnv`. The automaton `Dijkstra` describes the mutual exclusion algorithm designed by Dijkstra [Lyn96]. The automaton `DijkstraInt` is a simpler version of Dijkstra's algorithm that abstracts from those parts in the original algorithm dedicated to dealing with liveness. In other words, we have an abstraction hierarchy where the automata `MutexEnv`, `DijkstraInt` and `Dijkstra` lie respectively at the top, intermediate and lowest levels.

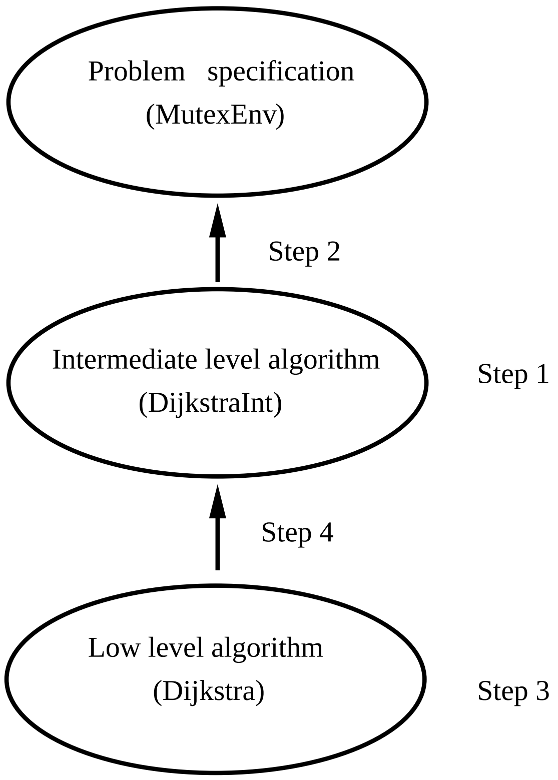
Figure 8 summarizes how we proceed in the rest of this section. We first present the IOA code for the intermediate level algorithm and use the IOA simulator to check whether it works as expected. To increase our confidence that it complies with the specification of mutual exclusion, we propose a forward simulation relation from `DijkstraInt` to `MutexEnv`. We then use the paired simulator to check that the proposed relation holds for the selected executions. We follow a similar line of action for the lower level algorithm. In this case we propose and check a forward simulation relation from `Dijkstra` to `DijkstraInt`. We know by Theorem 2.1 that if there is a forward simulation from `DijkstraInt` to `MutexEnv` and from `Dijkstra` to `DijkstraInt`, then *traces*(`Dijkstra`) must be a subset of *traces*(`MutexEnv`). That is to say all observable behaviors of `Dijkstra` are a subset of observable behaviors of `MutexEnv` and therefore satisfy mutual exclusion.

5.3.1 Intermediate level algorithm

The following is an IOA program which includes the description of the intermediate level algorithm and a schedule block to simulate the automaton `DijkstraInt`.

```
axioms NonDet
type Index = enumeration of p1, p2, p3
type Region = enumeration of rem, try, crit, exit
type PcValue = enumeration of rem, setflag01, setflag2, check, leavetry,
                              crit, reset, leaveexit
type Stage = enumeration of stage01, stage2

automaton DijkstraInt
  signature
    output try(p: Index)
    output crit(p: Index)
    output exit(p: Index)
    output rem(p: Index)
    internal setflag01(p: Index)
    internal setflag2(p: Index)
    internal check(p: Index, u: Index)
    internal reset(p: Index)
  states
    flag: Array[Index, Stage] := constant(stage01),
    pc: Array[Index, PcValue] := constant(rem),
    S: Array[Index, Set[Index]] := constant({})
  transitions
    output try(p)
      pre pc[p] = rem
      eff pc[p] := setflag01
    internal setflag01(p)
      pre pc[p] = setflag01
      eff flag[p] := stage01;
          pc[p] := setflag2
    internal setflag2(p)
      pre pc[p] = setflag2
```



Step 1: single-automaton simulation with `sim`
Step 2: paired simulation with `psim`
Step 3: single-automaton simulation with `sim`
Step 4: paired simulation with `psim`

Figure 8: Abstraction hierarchy

```
      eff flag[p] := stage2;
          S[p] := {p};
          pc[p] := check
    internal check(p, u)
      pre pc[p] = check ∧ ¬(u ∈ S[p])
      eff if flag[u] = stage2 then S[p] := {};
                                   pc[p] := setflag01
           else S[p] := S[p] ∪ {u};
                if size(S[p]) = 3 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] := stage01;
          S[p] := {};
          pc[p] := leaveexit
    output rem(p)
      pre pc[p] = leaveexit
      eff pc[p] := rem
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] = setflag01 then fire internal setflag01(p)
  elseif pc[p] = setflag2 then fire internal setflag2(p)
  elseif pc[p] = check then if ¬(p1 ∈ S[p]) then fire internal check(p,p1)
                            elseif ¬(p2 ∈ S[p]) then fire internal check(p,p2)
                            elseif ¬(p3 ∈ S[p]) then fire internal check(p,p3) fi
  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
```

Explanation of code The automaton DijkstraInt makes use of the types `PcValue` and `Stage` in addition to those that we have already introduced. The values of type `PcValue` represent the possible program counter values for the process while values of type `Stage` represent the stages of the algorithm. The phrase **axioms** `NonDet` is included to allow the use of operations specified by the trait `NonDet`.

The signature of `DijkstraInt` has three internal actions along with those of `MutexEnv`. It also has some state variables which are not present in `MutexEnv`. The algorithm specified by `DijkstraInt` has two stages. The first stage `stage01` indicates that a process is either inactive or is about to enter the second stage. The second stage `stage2` embodies the crucial steps and determines whether a process is allowed to enter the its critical region. A process can enter its critical region only if all other processes are in the first stage of the algorithm. The transition definition for action `check` details how this is checked. The state variables `flag` and `pc` are used respectively to record the stage of the algorithm for each process and to control the order of occurrence of the actions mimicking the program counter of a process. The schedule block implements a randomized scheduling policy for three processes. One of the three processes is picked randomly each time the while loop is executed. When `pc[p]` is `check` then the schedule block decides the process to be checked by `p`, by looking at `S[p]` and yielding the process with the smallest identifier that is not already in $S[p]$. Such a process is guaranteed to exist because `pc[p]` is no longer `check` once $S[p]$ contains all processes.

5.3.2 Running the simulator with `sim`

To simulate your code with `sim`:

1. Place your code in a file with extension .ioa, for example DijkstraInt.ioa
2. Check the code for syntax and semantic errors with `ioaCheck`
3. At the command line type
   $>$ `sim 100 DijkstraInt.ioa`
   where the first argument to `sim` is the number of required simulation steps and the second argument is the source file. The choice of number 100 here is arbitrary.

A sample output is presented in Section A.4 of the Appendix.
The following is the list of options available for running `sim`.

```
Usage
    sim [option] <# steps> [<automaton name>] <IL filename>

Options
  [-big]              Use BigInteger and BigReal for all calculations
  [-config <string>]+ Use the given configuration file(s) for options
  [-daikon]           Turn on Daikon instrumentation on
  [-dbg <string>]+    Turn on debug information for a java class or package.
  [-debug]            Turn on debug information globally
  [-ignoreFirst]      Ignore first program point (init states) during Daikon instrumentation
  [-noIl]             Do not send il output to a file (if reading an IOA file)
  [-o <string>]       Set base name for output
  [-odecls <string>]  Set destination file for decls output
  [-odtrace <string>] Set destination file for dtrace output
  [-oil <string>]     Set destination for il output
  [-rseed <number>]   Set randomizer seed for regression resting
  [-state]            Show all state variables during execution
  [-traces]           Show only traces during execution
  [-tracesOnly]       Show only traces during execution
```

5.3.3 Forward simulation from DijkstraInt to MutexEnv

The code below 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 5.3.1, this block allows one to use the paired simulator to check whether the relation holds in the simulated executions.

```
forward simulation from DijkstraInt to MutexEnv :
  ∀ i: Index (DijkstraInt.pc[i] = setflag01 ∨ DijkstraInt.pc[i] = setflag2 ∨
              DijkstraInt.pc[i] = check ∨ DijkstraInt.pc[i] = leavetry
                ⇔ MutexEnv.regionMap[i] = try);
  ∀ i: Index (DijkstraInt.pc[i] = crit ⇔ MutexEnv.regionMap[i] = crit);
  ∀ i: Index (DijkstraInt.pc[i] = rem ⇔ MutexEnv.regionMap[i] = rem);
  ∀ i: Index (DijkstraInt.pc[i] = reset ∨ DijkstraInt.pc[i] = leaveexit
                ⇔ MutexEnv.regionMap[i] = exit);
proof
    initially MutexEnv.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 setflag01(p:Index) ignore
    for internal setflag2(p:Index) ignore
    for internal check(p:Index,u:Index) ignore
    for internal reset(p:Index) ignore
```

Explanation of code 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 `setflag01`, `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.

5.3.4 Running the paired simulator with `psim`

To run the paired simulator:

1. Place the code is in a file with extension .ioa, for example InttoMutex.ioa

2. Check the code for syntax and semantic errors with `ioaCheck`

3. At the command line type
   `> psim 100 DijkstraInt MutexEnv InttoMutex.ioa`
   where the first argument to `psim` is the number of simulation steps, the second argument is the name of the low-level (implementation) automaton and the third argument is the name of the high-level (specification) automaton and the fourth one is the name of the source file. The choice for number 100 in this example is arbitrary.

A sample output is presented in Section A.5 of the Appendix.

The following is the list of options available for running `psim`.

```
Usage
   sim  [option] <numSteps> <implAut> <specAut> <filename>

Options
  [-big]              Use BigInteger and BigReal for all calculations
  [-config <string>]+ Use the given configuration file(s) for options
  [-daikon]           Turn on Daikon instrumentation on
  [-dbg <string>]+    Turn on debug information for a java class or package.
  [-debug]            Turn on debug information globally
  [-ignoreFirst]      Ignore first program point (init states) during Daikon instrumentation
  [-noIl]             Do not send il output to a file (if reading an IOA file)
  [-o <string>]       Set base name for output
  [-odecls <string>]  Set destination file for decls output
  [-odtrace <string>] Set destination file for dtrace output
  [-oil <string>]     Set destination for il output
  [-rseed <number>]   Set randomizer seed for regression resting
  [-state]            Show all state variables during execution
  [-traces]           Show only traces during execution
  [-tracesOnly]       Show only traces during execution
```

5.3.5 Forward simulation from `Dijkstra` to `DijkstaInt`

In this section we present the IOA code written for use with the paired simulator on automata `Dijkstra` and `DijkstaInt`. Note that the low-level automaton `Dijkstra` is presented for the first time. We do not explain it in detail as it is similar in many aspects to `DijkstraInt`. The main difference is that `Dijkstra` has three stages as opposed to two in `Dijkstra`. The additional stage is necessary to deal with the `turn` variable whose purpose is to guarantee that a process eventually enters its critical region. The internal actions which are present in `Dijkstra` but not in `DijkstraInt` all deal with testing and setting the variable `turn`.

```
type PcValueLow = enumeration of rem, setflag1, testturn, testflag, setturn,
                                 setflag2, check, leavetry, crit, reset,
                                 leaveexit
type StageLow = enumeration of stage0, stage1, stage2

automaton Dijkstra
 signature
    output try(p:Index)
    output crit(p:Index)
    output exit(p:Index)
    output rem(p:Index)
    internal setflag1(p: Index)
    internal setflag2(p: Index)
    internal testturn(p: Index)
    internal testflag(p, u: Index)
    internal setturn(p: Index)
    internal check(p: Index, u: Index)
    internal reset(p :Index)
 states
    turn: Index,
    flag: Array[Index, StageLow] := constant(stage0),
    pc: Array[Index, PcValueLow] := constant(rem),
    whose_flag: Array[Index, Index],
    S: Array[Index, Set[Index]] := constant({})
 transitions
    output try(p: Index)
      pre pc[p] = rem
      eff pc[p] := setflag1
    internal setflag1(p: Index)
      pre pc[p] = setflag1
      eff flag[p] := stage1;
          pc[p] := testturn
    internal testturn(p: Index)
      pre pc[p] = testturn
      eff if turn = p then pc[p] := setflag2
          else pc[p] := testflag;
               whose_flag[p] := turn
          fi
    internal testflag(p, u: Index)
      pre pc[p] = testflag ∧ whose_flag[p] = u
      eff if flag[u] = stage0 then pc[p] := setturn
          else pc[p] := testturn
          fi
    internal setturn(p: Index)
      pre pc[p] = setturn
      eff turn := p;
          pc[p] := setflag2
    internal setflag2(p: Index)
      pre pc[p] = setflag2
      eff flag[p] := stage2;
          S[p] := {p};
          pc[p] := check
    internal check(p, u: Index)
      pre pc[p] = check ∧ ¬(u ∈ S[p])
      eff if flag[u] = stage2 then S[p] := {};
                                   pc[p] := setflag1
          else S[p] := S[p] ∪ {u};
               if size(S[p]) = 3 then pc[p] := leavetry fi
          fi
    output crit(p: Index)
      pre pc[p] = leavetry
      eff pc[p] := crit
    output exit(p: Index)
      pre pc[p] = crit
      eff pc[p] := reset
    internal reset(p: Index)
      pre pc[p] = reset
      eff flag[p] := stage0;
          S[p] := {};
          pc[p] := leaveexit
    output rem(p: Index)
      pre pc[p] = leaveexit
      eff pc[p] := rem

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] = testturn then fire internal testturn(p)
    elseif (pc[p] = testflag ∧ whose_flag[p] ≠ p) then
                              fire internal testflag(p,whose_flag[p])
    elseif pc[p] = setturn then fire internal setturn(p)
    elseif pc[p] = setflag2 then fire internal setflag2(p)
    elseif pc[p] = check then if ¬(p1 ∈ S[p]) then fire internal check(p,p1)
                              elseif ¬(p2 ∈ S[p]) then fire internal check(p,p2)
                              elseif ¬(p3 ∈ S[p]) then fire internal check(p,p3)
                              fi
    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

forward simulation from Dijkstra to DijkstraInt :
    (Dijkstra.S = DijkstraInt.S);
    ∀ p:Index (Dijkstra.flag[p] = stage0 ∨ Dijkstra.flag[p] = stage1
                      ⇔ DijkstraInt.flag[p] = stage01);
    ∀ p:Index (Dijkstra.flag[p] = stage2 ⇔ DijkstraInt.flag[p] = stage2);
    ∀ p:Index (Dijkstra.pc[p] = rem ⇔ DijkstraInt.pc[p] = rem);
    ∀ p:Index (Dijkstra.pc[p] = setflag1 ⇔ DijkstraInt.pc[p] = setflag01);
    ∀ p:Index (Dijkstra.pc[p] = testturn ∨ Dijkstra.pc[p] = testflag ∨
                     Dijkstra.pc[p] = setturn ∨ Dijkstra.pc[p] = setflag2
                       ⇔ DijkstraInt.pc[p] = setflag2);
    ∀ p:Index (Dijkstra.pc[p] = check ⇔ DijkstraInt.pc[p] = check);
    ∀ p:Index (Dijkstra.pc[p] = leavetry ⇔ DijkstraInt.pc[p] = leavetry);
    ∀ p:Index (Dijkstra.pc[p] = crit ⇔ DijkstraInt.pc[p] = crit);
    ∀ p:Index (Dijkstra.pc[p] = reset ⇔ DijkstraInt.pc[p] = reset);
    ∀ p:Index (Dijkstra.pc[p] = leaveexit ⇔ DijkstraInt.pc[p] = leaveexit);

proof
     initially
     DijkstraInt.flag := constant(stage01);
     DijkstraInt.pc := constant(rem);
     DijkstraInt.S := constant({})
     for output try(p:Index) do fire output try(p) od
     for internal setflag1(p:Index) do fire internal setflag01(p) od
     for internal testturn(p:Index) ignore
     for internal testflag(p,u:Index) ignore
     for internal setturn(p:Index) ignore
     for internal setflag2(p:Index) do fire internal setflag2(p) od
     for internal check(p,u:Index) do fire internal check(p,u) od
     for output crit(p:Index) do fire output crit(p) od
     for output exit(p:Index) do fire output exit(p) od
     for internal reset(p:Index) do fire internal reset(p) od
     for output rem(p:Index) do fire output rem(p) od
```

Explanation of code The forward simulation relation is based on the idea that the first two stages (`stage0` and `stage1`) of algorithm `Dijkstra` are represented by a single stage in `DijkstraInt` (`stage01`). The rest of the code should be self-explanatory. The paired simulation can be carried out by placing the code for `DijkstraInt` from Section 5.3.1 in the same file as the code for `Dijkstra` with the schedule block and the proposed simulation relation.

6 Simulator-related extensions to the IOA language

In this section we revisit those parts of the IOA language that were modified in order accommodate the language constructs on which the IOA simulator depends. The modifications to the IOA syntax are described formally using a BNF grammar. We also comment on the semantic constraints for the extensions to the IOA language. The reader is referred to [GLV01] for the rest of the IOA grammar, the grammar syntax conventions used here and the semantics of the IOA Language.

6.1 Resolution of nondeterminism

As explained in Section 3, our approach to resolution of nondeterminism requires programmers to specify how the nondeterminism in an automaton is to be resolved by the simulator. The necessary modification to the IOA Language has two parts:

1. Addition of syntax for sequential programs that specify the values to choose or the transitions to schedule ("NDR programs").
2. Extensions to the existing syntax for **automaton** and **choose** that incorporate these sequential programs.

The resulting grammar is very similar to the existing `program` grammar in IOA, except that it permits the new **fire** and **yield** statements, used by the NDR mechanisms to schedule automaton actions and determine values of choices, as well as the **while** statement, which provides a looping construct with simple deterministic semantics.

Extension to primitive automaton syntax: This extension is straightforward: it simply provides a place to specify the schedule of a primitive automaton.

Original:

```
basicAutomaton      ::=  'signature' formalActions+ states transitions tasks?
```

Modified:

```
basicAutomaton      ::=  'signature' formalActions+ states transitions tasks? schedule?
schedule            ::=  'schedule' states? 'do' NDRProgram 'od'
NDRProgram          ::=  NDRStatement;*
NDRStatement        ::=  assignment
                       | NDRConditional
                       | NDRWhile
                       | NDRFire
NDRConditional      ::=  'if' predicate 'then' NDRProgram
                         ('elseif' predicate 'then' NDRProgram)*
                         ('else' NDRProgram)?  'fi'
NDRWhile            ::=  'while' predicate 'do' NDRProgram 'od'
NDRFire             ::=  'fire' actionType actionName actionActuals? transCase?
                       | 'fire'
```

An `assignment` in a schedule block may assign a value to any of the schedule's state variables, but it may not assign values to variables inside the automaton. This constraint is verified during static checking.

Determining values within a choose: This extension is also mostly straightforward. Besides providing a place to hold the `NDRProgram`, however, it does two additional things: first, it specifies a shorthand notation for a (presumably) common form of choice determination, and second, it allows for a choose statement to specify a variable name without a constraining where predicate. This is necessary for paired simulation, since the names of the chosen values in the specification automaton are still necessary to carry out the step correspondence, even in the absence of a where predicate.

Original:

```
choice              ::=  'choose' (variable 'where' predicate)?
```

Modified:

```
choice              ::=  'choose' (variable ('where' predicate)?)? choiceNDR?
choiceNDR           ::=  'det' 'do' NDRProgramY 'od'
                       | NDRYield
NDRProgramY         ::=  NDRStatementY;*
NDRStatementY       ::=  assignment
                       | NDRConditionalY
                       | NDRWhileY
                       | NDRYield
NDRConditionalY     ::=  'if' predicate 'then' NDRProgramY
                         ('elseif' predicate 'then' NDRProgramY)*
                         ('else' NDRProgramY)? 'fi'
NDRWhileY           ::=  'while' predicate 'do' NDRProgramY 'od'
NDRYield            ::=  'yield' term
```

The only statements appearing in a **yield** context are those that return values; specifically **fire** statements are disallowed.

6.2 Labeling transition definitions

As explained in Section 3, our approach to resolution of nondeterminism requires a way to refer to a transition definition in a primitive automaton. In general, it is not enough for this to specify the name and parameters of the transition: it is possible for two transitions with identical signature and where clause to be enabled in the same state. This addition to the IOA syntax remedies the situation by providing an explicit naming mechanism:

Original:

```
transition          ::=  actionHead chooseFormals?  precondition?  effect?
actionHead          ::=  actionType actionName (actionActuals where?)?
```

Modified:

```
transition          ::=  actionHead chooseFormals?  precondition?  effect?
actionHead          ::=  actionType actionName (actionActuals where?)?
                         transCase?
transCase           ::=  'case' idOrNumeral
```

The user is free to define, for a given action, two transitions with the same parameters and **case** name. The semantic checker does not issue an error message unless a **schedule** block for the automaton refers to such a duplicate transition. In case a duplicate transition is referred to, it indicates that more than one transition matches the given description, just as it would if there were no **case** names given.

6.3 Labeling invariants

It is convenient for invariants to have a name, so that the simulator can refer to the specific invariant in case it fails. This was accomplished with the following grammar change, which allows any numeral or identifier to be given as the name for an invariant.

Original:

```
invariant          ::=  'invariant' 'of' automatonName ':'  predicate
```

Modified:

```
invariant          ::=  'invariant' idOrNumeral?  'of' automatonName ':'  predicate
```

Because invariant labels exist only for the user's convenience in reading the simulator's output, the user is free to choose any (alphanumeric) name desired; no semantic checks are performed. For example, the user may give all invariants of an automaton the same name — this is considered as legal although it should obviously be avoided.

6.4 Paired simulation

In addition to the mathematical statement of a simulation relation between automata, the simulator also needs a step correspondence between the automata which realizes the simulation relation. Hence, it was necessary to develop a language for specifying these correspondences. See Section 4 for the semantics of this language, and for justification of the approach and terminology.

The syntax of IOA has been extended with **forward simulations** to permit the specification of a "proof", which embodies the step correspondence. This proof specifies, for each transition that the implementation automaton might take, a way to produce a sequence of transitions for the specification automaton. The following are the additions:

Original:

```
simulation         ::=  ('forward' | 'backward') 'simulation' 'from'
                        automatonName 'to' automatonName ':'  predicate
```

Modified:

```
simulation         ::=  ('forward' | 'backward') 'simulation' 'from'
                        automatonName 'to' automatonName ':'  predicate
                        simProof?
simProof           ::=  'proof' states?  ('initially' (variable ':=' term);+)?
                        simProofEntry+
simProofEntry      ::=  'for' actionType actionName
                        actionFormals?  transCase?
                        (('do' simProofProgram 'od') | 'ignore')
simProofProgram    ::=  simProofStatement;+
simProofStatement   ::=  assignment
                        | simProofConditional
                        | simProofWhile
                        | simProofFire
simProofConditional ::=  'if' predicate 'then' simProofProgram
                         ('elseif' predicate 'then' simProofProgram)*
                         ('else' simProofProgram)?  'fi'
simProofWhile       ::=  'while' predicate 'do' simProofProgram 'od'
simProofFire        ::=  'fire' actionType actionName
                         actionActuals?  transCase?
                         ('using' ( term 'for' variable ),+)?
```

The left-hand side of an assignment in a `simProofInit` block must refer to a state variable of the specification automaton. The user assumes the burden of ensuring that the `initially` assignments result in a reachable state of the specification automaton.

7 Implementation of the simulator

7.1 The IOA toolkit architecture

The simulator is part of the IOA toolkit, which is written in Java. The toolkit is split into two parts: the front end and the back end. The front end includes the IOA parser and syntax checker, while the back end includes the simulator, a code generator (to Java) and a translator to LSL. The tools share many components, and the shared parts are designed to facilitate adding new tools with minimal effort. The components can be divided into three categories:

- **Intermediate language and syntax trees** All back end tools use the same syntax tree to represent the IOA language structures as Java data structures. The front end generates an intermediate language (IL) representation of IOA, and back end libraries parse this IL into the shared syntax tree.[3]

- **Data structures for executable IOA** In addition to the simulator, the IOA code generator can also execute IOA programs[4]. To prevent redundant code and to ensure similar behavior, the toolkit programs that can execute IOA all use the same Java package for IOA data structures and functions.

- **Shared utility components** To provide similar behavior across all the IOA tools, many user interface and other features are implemented in shared libraries. In addition to the IL parser and syntax tree described below, the tools share an error handling mechanism, a command line argument processor and debug output generator.

Of course, all the tools are different in the ways they work with IOA. Specifically, some tools require only a subset of the language. For example, the simulator has no need for `assert` clauses in LSL specifications for data structures, but requires `schedule` and `det` blocks for nondeterminism resolution. In contrast, the the translator to LSL needs the `assert` clauses, but does not need nondeterminism resolution. We use the following rules for handling these implementation issues:

[3]Note that the front end has to have a syntax tree to parse IOA, but this tree is different from the back end tree, and does not interact with the back end. We shall henceforth call the front end parser the IOA parser and the back end IL parser the IL parser.

[4]Using the same ideas for nondeterminism resolution and scheduling that are presented in this paper

- The IOA front end parser understands all extensions of the language and writes IL files containing all the relevant information.
- The IL parser understands the core part of IOA, such as automaton signature, state variables and transition definitions.
- For a language structure that is specific to a particular tool, the tool is responsible for parsing and creating syntax trees for the structure.

The advantage of the above rules is that it makes the tools more independent from each other and the IL more robust to changes. The disadvantage is that implementing global features (like unparsing) is more difficult with respect to syntax trees.

7.2 The Intermediate language and IL parser

The IL is written to a text file by the IOA parser after an IOA file is read. It is meant to be "self contained": unlike an IOA file, it does not refer to external definitions such as LSL traits.

The format of the IL is parenthesized symbolic expressions (S-expressions), which are easily parsed and allow human reading and editing for debugging.

The convention is that the IOA parser and the IL parser do not write/read directly to/from text format. Instead, they parse/unparse the into S-expressions and then let a utility write to text form. This separates the steps involved in text processing and low-level parsing from the high-level recognition of IOA syntax structures. Another advantage of this is that the formatting and appearance of IL is the same when it is being generated by the IOA parser or the IL parser[5]. Lastly, when the IL parser finds an error in the IL, it uses the error handler common to all tools.

7.2.1 The `spec` object

Every IL file contains a top level object called the `spec`:

```
(ioa *sort-table* *operator-table* *variable-table*
      *automaton-definition* ...
      *annotations*)
```

The `spec` is a an S-expression list (S-list) that begins with the word `ioa`, and contains the symbol tables (one for data type sorts, one for operators and one for variable names), followed by the automaton definitions (more than one automaton can be defined), followed by any additional annotations for the spec.

The IL defines specific places where tool-specific extensions of the language may be placed: they are always at the end of S-lists and are written as S-lists following globally-recognized elements. The IL Parser delegates the parsing of tool-specific extensions back to the tool that invoked it.

[5]Even though they use different syntax trees, both of them generate S-expressions

7.2.2 Symbol tables

The IOA checker and parser resolve all name and scope issues, so that variables and operators share one flat namespace. The symbol tables map from this flat name space to the original IOA name space. The Simulator uses the flat namespace, but reports actions using the symbol table so users can refer to state variables and operators by their original names. For example in the following symbol tables:

```
(ioa
  (sorts                                    ;; *sort-table*
   (s0 "Bool" ())
   ...
   (s3 "Int" () lit)
   ...)
  (ops                                      ;; *operator-table*
   (op1 (infix "=") ((s0 s0) s0) (scope 0))
   ...
   (op452 (infix "=") ((s3 s3) s0) (scope 22))
   ...)
```

The operator `op452` is the `=` operator that operates on two arguments of type `Int` and returns a type `Bool`. Since the equality operator for integers is explicitly named, back end tools do not have to determine what a particular usage of `=` is. This is convenient because two data types may define and operator like `*` to mean different things (e.g. concatenation vs. multiplication).

7.2.3 Additional annotations

The two major types of annotations recognized by the shared IL parser in the `spec` object are shorthand sorts (such as tuple definitions) and invariant statements. Simulation relations between automata are annotations that are parsed only by the simulator and LSL generators.

7.2.4 Automaton definitions

Each automaton definition is an S-list that consists of a description of the actions, the state variables (and their initializations), the possible transitions followed by tool-specific annotations. The only annotation the simulator uses is a `schedule` block for nondeterminism resolution.

```
(automaton "Channel"
 ((actions
    (a0 input "send" (formals v1))
    (a1 output "receive" (formals v1)))
   (states *state-variables*)
   (transitions *transitions-list*)
   (schedule *schedule-block*)))
```

Laura Dean's thesis [Dea01] contains the formal BNF specification of the IL, along with the simulator-specific extensions.

7.3 Implementation of the IL

In this section, we briefly look at the way the IL syntax tree is implemented. For a more detailed view, see Ramirez's thesis [RR00].

Every object in the IL tree is a Java interface that inherits from `ioa.il.ILElement`. For example, `ioa.il.Program` is an `ioa.il.ILElement` that contains multiple `ioa.il.Statements`. Each of these interfaces is implemented with Java objects that inherit from `ioa.il.BasicILElement`. There are two reasons for using interfaces rather than objects for the IL:

- Tools can choose to implement the IL in a completely different way from the default objects under `ioa.il.BasicIlElement`.
- Java does not permit multiple inheritance in objects, so using interfaces provides more flexibility for tools that want to extend object functionality.

Each back-end tool can choose to directly use classes in `ioa.il` to implement its functionality, or it can extend some of the objects derived from `ioa.il.BasicILElement` and create a parallel syntax tree for itself. The convention is to delegate standard functionality to `ioa.il` objects whenever possible. Therefore , for example, `ioa.simulator.SimChoice` extends `ioa.il.NDRChoice` which extends `ioa.il.BasicValue`. `ioa.simulator.SimChoice` does not directly extend `ioa.simulator.SimValue` (which extends `ioa.il.BasicValue`).

To parse and generate IL tree objects, the factory design pattern is used. The ILParser is a subroutine called by back-end tools that does the actual parsing. ILParser generates objects in the tree as needed by asking an ILFactory. By default, the ILParser uses `ioa.il.BasicILFactory` which produces children of `ioa.il.BasicILElement`. Back-end tools that want to replace IL tree objects with customized ones just have to change the factory that is used to a custom one. The simulator thus uses a `ioa.simulator.SimILFactory`.

7.4 Simulator data types

The Simulator shares runtime type libraries with the IOA Code Generator to ensure similar code behavior and to reduce repeated code. The toolkit refers to these as abstract data types (ADTs) and Michael Tsai in "ADTs for IOA Code Generation" [Tsa01] describes the process in detail.

Data types and associated operators used in IOA are specified either explicitly (in LSL files) or implicitly (built in) to the IOA parser and checker. These specifications are implemented by ADTs in the runtime libraries. When an IOA program is run and an operator or data type is constructed in the IL tree, the Simulator looks up the appropriate implementation in an ADT "Registry" that maps operator and sort specifications to implementations. The implementation sort or operator is then used when working with data values.

7.4.1 The ADT registry

Before the Registry is used, it must be told which IOA operators and sorts are being implemented by what. This is done by a set of *registration classes*. For example, the registration class for `IntSort` tells the Registry that: the IOA data type `Int` will be implemented by the Java class `IntSort`, and the operators that work on Int (such as `+`) will be implemented by methods in `IntSort`.

A registration class may register for any number of operators or sorts, but the convention is to use one registration class for each IOA data type and its associated operators. For functions that operate on multiple sorts, registration can be done by any of the sorts' registration classes.

Since specifications are separate from implementation, users can choose to have an alternate set of data type implementations. This is done by configuring the Registry to use a different set of registration classes in the `.ioarc` configuration file.

It is important to note that with this flexible registration mechanism, mismatches in registration are not detected at compile time. For example, if an ADT was missing and a registration class referred to it, the registration class would still compile. Only when the simulator is run would this error be detected. This makes good testing and error checking vital (see below).

7.5 Testing and implementation

The IOA toolkit also shares testing infrastructure between its tools. There are two types of tests:

- **Unit tests** These test a few classes for their expected functionality by themselves. This is done using Junit[JUn02]. Currently, all the ADT implementations and some shared interface libraries are tested this way. Testing the ADTs with unit tests is important as it would be troublesome to generate IOA files that call every method in an ADT implementation.
- **Regression tests** All the output generated by IOA tools is compared to the expected output using a test suite of more than 30 tests. These tests check for correct implementation of IOA data and language structures, and each test is run for each tool.

Extensions to the Simulator or other tools should also add the appropriate unit and regression tests to ensure verification of correct operation.

A Simulator outputs

This section includes the simulator outputs for the examples presented throughout this paper. (Note: some of them need to be updated).

A.1 Simulator output for `Chooser`

```
[[[[ Begin initialization [[[[
%%%% Modified state variables:
     chosen --> 87
     did_choose --> false
]]]] End initialization ]]]]
[[[[ Begin step 1 [[[[
    transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 11
     did_choose --> true
]]]] End step 1 ]]]]
[[[[ Begin step 2 [[[[
    transition: output action2(11) in automaton Chooser
%%%% No modified state variables
]]]] End step 2 ]]]]
[[[[ Begin step 3 [[[[
    transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 12
     did_choose --> true
]]]] End step 3 ]]]]
[[[[ Begin step 4 [[[[
    transition: output action2(12) in automaton Chooser
%%%% No modified state variables
]]]] End step 4 ]]]]
[[[[ Begin step 5 [[[[
    transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 13
     did_choose --> true
]]]] End step 5 ]]]]

...


[[[[ Begin step 95 [[[[
    transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 13
     did_choose --> true
]]]] End step 95 ]]]]
[[[[ Begin step 96 [[[[
    transition: output action2(13) in automaton Chooser
%%%% No modified state variables
]]]] End step 96 ]]]]
[[[[ Begin step 97 [[[[
    transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 11
     did_choose --> true
]]]] End step 97 ]]]]
[[[[ Begin step 98 [[[[
    transition: output action2(11) in automaton Chooser
%%%% No modified state variables
]]]] End step 98 ]]]]
[[[[ Begin step 99 [[[[
    transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 12
     did_choose --> true
]]]] End step 99 ]]]]
[[[[ Begin step 100 [[[[
    transition: output action2(12) in automaton Chooser
%%%% No modified state variables
]]]] End step 100 ]]]]
No errors
```

A.2 Simulator output for `Fibonacci`

```
[[[[ Begin initialization [[[[
%%%% Modified state variables:
     a --> 1
     b --> 0
     c --> 1
]]]] End initialization ]]]]
[[[[ Begin step 1 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 0
     b --> 1
     c --> 1
>>>> Invariant B failed
]]]] End step 1 ]]]]
[[[[ Begin step 2 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 1
     b --> 1
     c --> 2
>>>> Invariant B failed
]]]] End step 2 ]]]]
[[[[ Begin step 3 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 1
     b --> 2
     c --> 3
>>>> Invariant B failed
]]]] End step 3 ]]]]
[[[[ Begin step 4 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 2
     b --> 3
     c --> 5
>>>> Invariant B failed
]]]] End step 4 ]]]]
[[[[ Begin step 5 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 3
     b --> 5
     c --> 8
>>>> Invariant B failed
]]]] End step 5 ]]]]
**** Some errors occured during simulation
```

A.3 Forward simulation from `FiniteGreeter` to `GreeterSpec`

```
[[[[ Begin initialization [[[[
%%%% Modified state variables for impl automaton:
     maxGreets --> 100
     count --> 0
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End initialization ]]]]
[[[[ Begin step 1 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 1
     Executed spec transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End step 1 ]]]]
[[[[ Begin step 2 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 2
     Executed spec transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End step 2 ]]]]
[[[[ Begin step 3 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 3
     Executed spec transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End step 3 ]]]]
[[[[ Begin step 4 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 4
     Executed spec transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End step 4 ]]]]

...

[[[[ Begin step 15 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 15
     Executed spec transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End step 15 ]]]]
[[[[ Begin step 16 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 16
     Executed spec transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End step 16 ]]]]

...

[[[[ Begin step 99 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 99
     Executed spec transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] End step 99 ]]]]
[[[[ Begin step 100 [[[[
     Executed impl transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 100
     Executed spec transition: output hello in automaton GreeterSpec using false for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> false
]]]] End step 100 ]]]]
>>>> No errors
```

A.4 Simulator output for `DijkstraInt`

```
[[[[ Begin initialization [[[[
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01))
     pc --> (ArraySort (ConstantValue rem))
     S --> (ArraySort (ConstantValue ()))
]]]] End initialization ]]]]
```

```
[[[[ Begin step 1 [[[[
    transition: output try(p3) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p3 setflag01))
]]]] End step 1 ]]]]
[[[[ Begin step 2 [[[[
    transition: output try(p2) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p2 setflag01) (p3 setflag01))
]]]] End step 2 ]]]]
[[[[ Begin step 3 [[[[
    transition: output try(p1) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag01) (p2 setflag01) (p3 setflag01))
]]]] End step 3 ]]]]
[[[[ Begin step 4 [[[[
    transition: internal setflag01(p1) in automaton DijkstraInt
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage01))
     pc --> (ArraySort (ConstantValue rem) (p1 setflag2) (p2 setflag01) (p3 setflag01))
]]]] End step 4 ]]]]

...

[[[[ Begin step 52 [[[[
    transition: internal setflag2(p2) in automaton DijkstraInt
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage2) (p2 stage2) (p3 stage2))
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 check) (p3 leavetry))
     S --> (ArraySort (ConstantValue ()) (p1 (p1 p2)) (p2 (p2)) (p3 (p1 p2 p3)))
]]]] End step 52 ]]]]
[[[[ Begin step 53 [[[[
    transition: output crit(p3) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 check) (p3 crit))
]]]] End step 53 ]]]]
[[[[ Begin step 54 [[[[
    transition: internal check(p2, p1) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 setflag01) (p3 crit))
     S --> (ArraySort (ConstantValue ()) (p1 (p1 p2)) (p2 ()) (p3 (p1 p2 p3)))
]]]] End step 54 ]]]]
[[[[ Begin step 55 [[[[
    transition: output exit(p3) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 setflag01) (p3 reset))
]]]] End step 55 ]]]]
[[[[ Begin step 56 [[[[
    transition: internal reset(p3) in automaton DijkstraInt
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage2) (p2 stage2) (p3 stage01))
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 setflag01) (p3 leaveexit))
     S --> (ArraySort (ConstantValue ()) (p1 (p1 p2)) (p2 ()) (p3 ()))
]]]] End step 56 ]]]]
```

```
[[[[ Begin step 57 [[[[
    transition: internal check(p1, p3) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 leavetry) (p2 setflag01) (p3 leaveexit))
     S --> (ArraySort (ConstantValue ()) (p1 (p1 p2 p3)) (p2 ()) (p3 ()))
]]]] End step 57 ]]]]

....

[[[[ Begin step 62 [[[[
    transition: output crit(p1) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 leaveexit))
]]]] End step 62 ]]]]
[[[[ Begin step 63 [[[[
    transition: output rem(p3) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 rem))
]]]] End step 63 ]]]]
[[[[ Begin step 64 [[[[
    transition: internal setflag2(p2) in automaton DijkstraInt
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage2) (p2 stage2) (p3 stage01))
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 check) (p3 rem))
     S --> (ArraySort (ConstantValue ()) (p1 (p1 p2 p3)) (p2 (p2)) (p3 ()))
]]]] End step 64 ]]]]
[[[[ Begin step 65 [[[[
    transition: internal check(p2, p1) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag01) (p3 rem))
     S --> (ArraySort (ConstantValue ()) (p1 (p1 p2 p3)) (p2 ()) (p3 ()))
]]]] End step 65 ]]]]
[[[[ Begin step 66 [[[[
    transition: internal setflag01(p2) in automaton DijkstraInt
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage2) (p2 stage01) (p3 stage01))
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 rem))
]]]] End step 66 ]]]]
[[[[ Begin step 67 [[[[
    transition: output try(p3) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 setflag01))
]]]] End step 67 ]]]]
[[[[ Begin step 68 [[[[
    transition: output exit(p1) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 reset) (p2 setflag2) (p3 setflag01))
]]]] End step 68 ]]]]
[[[[ Begin step 69 [[[[
    transition: internal reset(p1) in automaton DijkstraInt
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage01) (p2 stage01) (p3 stage01))
     pc --> (ArraySort (ConstantValue rem) (p1 leaveexit) (p2 setflag2) (p3 setflag01))
     S --> (ArraySort (ConstantValue ()) (p1 ()) (p2 ()) (p3 ()))
]]]] End step 69 ]]]]

...

[[[[ Begin step 81 [[[[
    transition: output crit(p3) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag2) (p2 setflag2) (p3 crit))
]]]] End step 81 ]]]]

...

[[[[ Begin step 100 [[[[
    transition: internal setflag2(p3) in automaton DijkstraInt
%%%% Modified state variables:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage2) (p2 stage2) (p3 stage2))
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 leavetry) (p3 check))
     S --> (ArraySort (ConstantValue ()) (p1 (p1)) (p2 (p1 p2 p3)) (p3 (p3)))
]]]] End step 100 ]]]]
No errors
```

A.5 Forward simulation from `DijkstraInt` to `MutexEnv`

```
[[[[ Begin initialization [[[[
%%%% Modified state variables for impl automaton:
     flag --> (ArraySort (ConstantValue stage01))
     pc --> (ArraySort (ConstantValue rem))
     S --> (ArraySort (ConstantValue ()))
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem))
]]]] End initialization ]]]]
[[[[ Begin step 1 [[[[
     Executed impl transition: output try(p2) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p2 setflag01))
     Executed spec transition: output try(p2) in automaton MutexEnv
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p2 try))
]]]] End step 1 ]]]]
[[[[ Begin step 2 [[[[
     Executed impl transition: output try(p1) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag01) (p2 setflag01))
     Executed spec transition: output try(p1) in automaton MutexEnv
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p1 try) (p2 try))
]]]] End step 2 ]]]]
[[[[ Begin step 3 [[[[
     Executed impl transition: output try(p3) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag01) (p2 setflag01) (p3 setflag01))
     Executed spec transition: output try(p3) in automaton MutexEnv
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p1 try) (p2 try) (p3 try))
]]]] End step 3 ]]]]

...

[[[[ Begin step 9 [[[[
     Executed impl transition: output crit(p2) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag2) (p2 crit) (p3 setflag01))
     Executed spec transition: output crit(p2) in automaton MutexEnv
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p1 try) (p2 crit) (p3 try))
]]]] End step 9 ]]]]

...

[[[[ Begin step 59 [[[[
     Executed impl transition: internal check(p2, p3) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 leavetry) (p3 setflag2))
     S --> (ArraySort (ConstantValue ()) (p1 (p1)) (p2 (p1 p2 p3)) (p3 ()))
]]]] End step 59 ]]]]
[[[[ Begin step 60 [[[[
     Executed impl transition: output crit(p2) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 check) (p2 crit) (p3 setflag2))
     Executed spec transition: output crit(p2) in automaton MutexEnv
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p1 try) (p2 crit) (p3 try))
]]]] End step 60 ]]]]
[[[[ Begin step 61 [[[[
     Executed impl transition: internal check(p1, p2) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag01) (p2 crit) (p3 setflag2))
     S --> (ArraySort (ConstantValue ()) (p1 ()) (p2 (p1 p2 p3)) (p3 ()))
]]]] End step 61 ]]]]
[[[[ Begin step 62 [[[[
     Executed impl transition: internal setflag01(p1) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage01) (p2 stage2) (p3 stage01))
     pc --> (ArraySort (ConstantValue rem) (p1 setflag2) (p2 crit) (p3 setflag2))
]]]] End step 62 ]]]]
[[[[ Begin step 63 [[[[
     Executed impl transition: output exit(p2) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag2) (p2 reset) (p3 setflag2))
     Executed spec transition: output exit(p2) in automaton MutexEnv
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p1 try) (p2 exit) (p3 try))
]]]] End step 63 ]]]]
[[[[ Begin step 64 [[[[
     Executed impl transition: internal setflag2(p3) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage01) (p2 stage2) (p3 stage2))
     pc --> (ArraySort (ConstantValue rem) (p1 setflag2) (p2 reset) (p3 check))
     S --> (ArraySort (ConstantValue ()) (p1 ()) (p2 (p1 p2 p3)) (p3 (p3)))
]]]] End step 64 ]]]]
[[[[ Begin step 65 [[[[
     Executed impl transition: internal reset(p2) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     flag --> (ArraySort (ConstantValue stage01) (p1 stage01) (p2 stage01) (p3 stage2))
     pc --> (ArraySort (ConstantValue rem) (p1 setflag2) (p2 leaveexit) (p3 check))
     S --> (ArraySort (ConstantValue ()) (p1 ()) (p2 ()) (p3 (p3)))
]]]] End step 65 ]]]]

...

[[[[ Begin step 100 [[[[
     Executed impl transition: internal check(p3, p1) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 setflag01) (p2 check) (p3 setflag01))
     S --> (ArraySort (ConstantValue ()) (p1 ()) (p2 (p2)) (p3 ()))
]]]] End step 100 ]]]]
>>>> No errors
```

B Trait NonDet

```
NonDet: trait
  introduces
    randomNat: Nat, Nat → Nat
         % uniformly random natural number in given range
    queryNat: Nat, Nat → Nat
         % query user for natural number in given range
    randomInt: Int, Int → Int
         % uniformly random integer in given range
    queryInt: Int, Int → Int
         % query user for integer in given range
    randomBool: → Bool
         % random boolean (each value with probability 0.5)
```

References

[Che98] A. E. Chefter. A simulator for the IOA language. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1998.

[Dea01] Laura G. Dean. Improved simulation of Input/Output automata. Master's thesis, Massachusetts Institute of Technology, 2001.

[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.

[GLV01] S. Garland, N. Lynch, and M. Vaziri. *IOA: A Language for Specifying, Programming, and Validating Distributed Systems.* MIT Laboratory for Computer Science, Cambridge, MA, 2001. URL `http://theory.lcs.mit.edu/tds/ioa.html`.

[JUn02] JUnit. Junit. `www.junit.org`, 2002.

[KCD+] D. Kaynar, A. Chefter, L. Dean, S. Garland, N. Lynch, T. Ne Win, and A. RamírezRobredo. Simulating nondeterministic systems at multiple-levels of abstraction. Submitted for publication.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Shaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[PAG] PAG. Home page of the Daikon invariant detector project. Maintained by the Program Analysis Group at MIT Laboratory for Computer Science (leader Michael Ernst). URL `http://pag.lcs.mit.edu/daikon/`.

[RR00] J. Antonio Ramırez-Robredo. Paired simulation of I/O automata. Master's thesis, Massachusetts Institute of Technology, 2000.

[TDS] TDS. Home page of the IOA project. Maintained by the Theory of Distributed Systems Group at MIT Laboratory for Computer Science (leader Nancy Lynch). URL `http://theory.lcs.mit.edu/tds/ioa.html`.

[Tsa01] Michael Tsai. Abstract data types for IOA code generation. Technical report, MIT Laboratory for Computer Science, 2001.

[Win02] Toh Ne Win. Assisting IOA design and verification with Daikon. Presentation Slides, March 2002.

[WS01] Toh Ne Win and Gustavo Santos. The IOA-Daikon connection: Enabling dynamic invariant discovery in IOA programs. Technical report, MIT Laboratory for Computer Science, 2001.