Improved Simulation of Input/Output Automata

by

Laura Dean

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author ....
Department of Electrical Engineering and Computer Science
September 10, 2001

Certified by ....
Michael D. Ernst
Assistant Professor
Thesis Supervisor

Certified by ....
Nancy A. Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Improved Simulation of Input/Output Automata

by

Laura Dean

Submitted to the Department of Electrical Engineering and Computer Science
on September 10, 2001, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The IOA Simulator is part of a collection of tools for developing and analyzing distributed algorithms. This thesis describes several improvements to the Simulator: adding new data type implementations, sharing data types with the IOA Code Generator, improving the Simulator's documentation, and adding an interface to Daikon, an invariant-discovery tool. These improvements should increase the Simulator's usability as a tool in writing and verifying algorithms for distributed systems.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Thesis Supervisor: Nancy A. Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

My advisors, Nancy Lynch and Michael Ernst, were extremely helpful, and their input greatly improved the quality of this thesis. Their advice and encouragement were invaluable.

Antonio Ramírez was always readily available to answer questions over email. Michael Tsai and Toh Ne Win were amazingly helpful in all aspects of IOA and Java programming. Additionally, it was Toh who suggested an IOA implementation of the Towers of Hanoi game, which appears in this thesis. Josh Tauber was great to talk to, both for technical IOA discussion and for sympathy about the process of writing a thesis proposal. Shien Jin Ong was an excellent tester and a source of great insight into what the Simulator should be able to do in the future. Steve Garland answered my questions about (and fixed critical bugs in) the IOA Front End.

Rachel Caileff, Sara Barron, Amy Gresser, and many other friends were essential in convincing me that I would eventually finish. I received similar encouragement from my thesis contemporaries: (Chris) Beland, Sarah McDougal, Susama Agarwala, David Maze, and Victor Luchangco, and from my office-mates. And of course, I could not have completed this work without the love and support of my family.

Contents

List of Listings

Chapter 1

Introduction

Distributed systems can be quite complicated, and it is therefore important to be able to reason about a system at both the high level and the implementation level. The I/O Automaton model [8, 9] encourages the programmer to practice a process of successive refinement, describing a system first at a high level, and then at lower levels, with increasing detail. The IOA language provides an expressive medium for precise description of a system's behavior, allows for descriptions at various levels of abstraction, and provides a mechanism for relating those descriptions.

The IOA Simulator allows the programmer to view possible executions of systems developed by this process, potentially detecting bugs or gaining confidence in the algorithm. Additionally, it can act as an interface to the Daikon invariant-detection system [4].

1.1 IOA tools

The IOA Toolkit includes several types of assistance for developing algorithms and programs. They are grouped into front-end, which take IOA input, and back-end tools, which take input in an intermediate language (IL). The first kind listed here are the front-end tools:

- The IOA Front End reports syntactic and semantic errors in IOA input; if it finds no errors, it produces output in the intermediate language (IL) understood by the back-end tools.

- The Composer, first described in Anna Chefter's thesis [3], will take descriptions of IOA automata that are designed to work together and combine them into a single automaton. Steve Garland and Joshua Tauber are currently writing a precise specification for the semantics of composition, and they will add the Composer to the IOA Front End.

The toolkit also includes the following back-end tools:

- The IL parser, written by Antonio Ramírez, generates a parse tree from the IL. It is used by all the other back-end tools.

- Translation tools provide interfaces to theorem provers and model checkers, to enable computer-assisted verification of programs. Andrej Bogdanov has created an IOA interface to the Larch Prover [7, 1]. Stanislav Funiak is working on automated translation of IOA to the TLA+ specification language, which is used by the TLC model checker [13].

- The Code Generator (CodeGen) will produce Java from IOA. This will allow a programmer to develop an algorithm in IOA, perhaps using the other tools to assist in verification, and then generate a distributed implementation directly. Tauber and Michael J. Tsai are developing CodeGen.

- The Simulator interprets IOA programs, producing a trace of a possible schedule of automaton transitions. This allows a programmer to observe possible executions of a program for debugging and testing. The Simulator is the focus of this thesis.

1.2 IOA and Simulator Background

The IOA Manual [6] begins with a tutorial on the IOA language; the tutorial includes mathematical definitions for I/O automata and IOA.

The IOA Simulator was designed by Anna Chefter [3] and implemented by Antonio Ramírez [11]. This section describes two problems — the nondeterminism of IOA and paired simulation — and the Simulator's approach to them.

1.2.1 The nondeterminism of IOA

IOA specifications are inherently nondeterministic. They do not provide an explicit list of commands in order; instead, they define automata in terms of their state variables and allowed transitions, and they define those transitions in terms of preconditions and effects. (These definitions may constrain the order of the transitions, of course.)

One source of nondeterminism in IOA specifications is that, in a given state, multiple transitions might be enabled. For example, transitions `t1` and `t2` might both be enabled, or transitions `t1(5)` and `t1(10)`. Both examples are cases of *scheduling nondeterminism.*

Another source of nondeterminism is IOA's `choose` statement, e.g. `choose i: Nat where i<5`, which indicates that i must be a natural number between 0 and 5. This is referred to as ***`choose`*** *nondeterminism.*

1.2.2 The current approach to resolving nondeterminism

Since the Simulator's job is to create an execution, it must resolve both `choose` and scheduling nondeterminism.

In order to resolve scheduling nondeterminism, an automaton specification must be accompanied by a `schedule` block, for all but the simplest automata. The precise definition of "all but the simplest" appears in Section 2.4. The user must also provide explicit instructions for resolving `choose` nondeterminism. Sections 2.5 and 2.6 provide examples of these kinds of nondeterminism and how to resolve them.

1.2.3 Paired simulation

When a low-level automaton is an acceptable implementation of a high-level goal, the automata exhibit a property called *trace inclusion*: an automaton `Impl` implements an automaton `Spec` if any possible trace of an execution of `Impl` is also a possible trace of an execution of `Spec`. A *trace* of a transition or an execution is a sequence, possibly empty, of its input and output transitions. `Spec` and `Impl` are referred to as the *specification automaton* and the *implementation automaton*, respectively.

A *simulation relation* from `Impl` to `Spec` implies trace inclusion; this property is what makes simulation relations important enough to be reflected in the IOA language. If a simulation relation holds between the states of `Impl` and `Spec`, then for any transition x of `Impl`, there is a sequence α of transitions of `Spec` such that:

- the traces of x and α are identical, and
- the simulation relation holds between the states of `Impl` and `Spec` after x and α have occurred.

Section 2.7 includes examples of simulation relations in IOA and how the Simulator can use them.

The following, more precise, definition is taken from Lynch [8]: A forward simulation relation from automaton A to automaton B is a binary relation $f \subseteq \mathit{states}(A) \times \mathit{states}(B)$ such that:

1. If $s \in \mathit{start}(A)$, then $f(s) \cap \mathit{start}(B) \neq \emptyset$;
2. if s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and (s, π, s') is a transition of A, with pre-state s and post-state s', then there is an execution fragment α of B starting with state u and ending with some state $u' \in f(s')$, such that $\mathit{trace}(\alpha) = \mathit{trace}(\pi)$,

where $\mathit{start}(A)$ is the set of start states of A, and $f(s)$ stands for $\{u : (s, u) \in f\}$.

1.3 Daikon

Daikon is an invariant-detection tool developed by Michael Ernst and colleagues [4]. It deduces potential invariants based on information about given executions of a program. It reports only facts that are true in every execution it sees; this does not constitute proof of an invariant, but the information is often interesting and useful.

Daikon has interfaces for Java, C/C++, and now IOA. This thesis describes the development of the Simulator's ability to create output that Daikon can read; Toh Ne Win is continuing work on this functionality.

1.4 Outline of this thesis

Chapter 2 is a user's guide to the Simulator. It describes what the Simulator can and cannot do, with IOA program examples to illustrate its capabilities. Most of the features described in the guide were implemented by Ramírez, though some are new; the new features are explicitly identified as such.

Chapter 3 describes the Simulator's ability to produce output for Daikon.

Chapter 4 describes the current state of the syntax and semantics of Simulator-related extensions to the IOA language. These extensions include nondeterminism resolution mechanisms and "proof" blocks, which tell the Simulator how to run the the specification automaton in a paired simulation.

Chapter 5 contains notes on the IOA intermediate language (IL). This chapter is of interest to Simulator developers, but not to casual users.

Chapter 6 contains notes on the implementation of the Simulator. Like Chapter 5, it is aimed at Simulator developers.

Chapter 7, the final chapter, contains a conclusion and suggestions for future work.

1.5 Terminology

The word "specification" has two meanings in this thesis. As a noun, as in "this IOA specification has correct syntax, but contains a semantic error," it refers to IOA code specifying an automaton. As an adjective, as in "the specification automaton is named `Fibonacci`," it refers to one of the automata in a paired simulation.

"IOA specification" also contrasts with "IOA program." An IOA program includes information on how the Simulator should resolve the automaton's nondeterminism. The distinction can be viewed as a subset relation, because every program is also a specification; in this thesis, the word "program" will be used whenever it is applicable.

"Simulation" is another dual-use word. It can refer to the action the Simulator performs on an automaton (simulating it), or it can refer to a simulation relation between an implementation and a specification automaton.

Chapter 2

User's Guide

This chapter introduces the Simulator to a reader with some familiarity with I/O automata and IOA. The first part of the IOA Manual [6] is a good source of the required background. For those who prefer a less mathematical introduction, it may be helpful to read the examples here first, or in parallel.

The chapter contains a brief introduction to the Simulator, a series of example IOA programs, and additional details about programs the Simulator can and cannot run. The series of examples begins with a simple deterministic automaton and then proceeds to more complicated examples in order to demonstrate the Simulator's capabilities.

2.1 The purpose of the Simulator

As described in Section 1.2, the Simulator creates executions of IOA automata. This capability makes it useful as a tool for testing automata, because each execution either reveals bugs or increases confidence that an automaton works as expected, by displaying values of state variables and verifying that invariants hold. Additionally, the Simulator serves as an IOA interface to the Daikon invariant-discovery tool.

2.2 A simple deterministic example

The `Fibonacci` specification in Listing 2-1 defines a simple deterministic automaton: it has no `choose` statements, it has only one transition, and that transition has no parameters. The effect of the `compute` transition is to compute the next Fibonacci number and store it as c. At every step, the Simulator runs the `compute` transition, since that is its only available option. It then checks that the stated invariant holds. At the nth step, the state variable c holds the value of the nth Fibonacci number. Listing 2-2 shows the Simulator's output for `sim 6 Fibonacci.ioa`.

Listing 2-1: `Fibonacci.ioa`

```
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 A of Fibonacci:
  a + b = c
```

Listing 2-2: Simulator output for `sim 6 Fibonacci.ioa`

```
[[[[ 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
]]]] End step 1 ]]]]
[[[[ Begin step 2 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 1
     b --> 1
     c --> 2
]]]] End step 2 ]]]]
[[[[ Begin step 3 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 1
```

```
     b --> 2
     c --> 3
]]]] End step 3 ]]]]
[[[[ Begin step 4 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 2
     b --> 3
     c --> 5
]]]] End step 4 ]]]]
[[[[ Begin step 5 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 3
     b --> 5
     c --> 8
]]]] End step 5 ]]]]
[[[[ Begin step 6 [[[[
    transition: internal compute in automaton Fibonacci
%%%% Modified state variables:
     a --> 5
     b --> 8
     c --> 13
]]]] End step 6 ]]]]
```

2.3 How to run the Simulator

The IOA Toolkit distribution includes a jar file with all the necessary Java classes for running the IOA tools. The toolkit's `bin` directory includes several scripts for running tools, including `sim` and `psim`, short for "simulator" and "paired simulator." Running these programs requires Java version 1.3 or later.

The toolkit also includes the source code for the tools, in its `Code` directory. Running `make` there builds everything; running `make` in the `bin` directory rebuilds the jar file and the scripts for running the programs. Our group compiles the toolkit with JDK 1.3 and PolyJ 1.0.2; later versions of these compilers may work as well, but earlier versions will not.

2.3.1 Command-line options

The simulator and the paired simulator are called as follows.

```
sim [flags] <numSteps> [<automaton name>] <filename>
```

```
psim [flags] <numSteps> <implAut> <specAut> <filename>
```

The input file should be in IOA or in the intermediate language (IL). If that file specifies only one automaton, then `sim` will deduce that it is the automaton to be simulated; otherwise, it must be explicitly stated on the command line. The following table lists the optional flags and describes them. Flags marked with an asterisk are expected to be used only by Simulator developers and maintainers.

Flag	Effect
-daikon	Turn on Daikon instrumentation.
-dbg <string>+*	Turn on debug information for the named Java classes or packages.
-debug *	Turn on debug information globally.
-ignoreFirst	Do not print information to the dtrace file until the first transition has occurred, during Daikon instrumentation. (An example of using this option appears in Section 3.6.)
-noIl	Do not send IL output to a file (if input is an IOA file).
-o <string>	Set base name for output files to <string>.
-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 testing).
-state	Show all state variables during execution.
-traces (or -tracesOnly)	Show only traces (no state variables) during execution. (By default, all modified state variables are shown.)

2.4 Simulator requirements

This section describes various aspects of what the Simulator requires in order to run an IOA program.

2.4.1 Resolving nondeterminism

`Fibonacci` as shown in Listing 2-1 is a simple deterministic automaton. It defines only one transition: `compute`, with no parameters. These properties make it easy for the Simulator to know how to run it — at each step, the Simulator simply performs the `compute` transition.

If an automaton has two or more transition definitions, but the transitions do not take any parameters, then the Simulator can still run the automaton with no additional information. At each step, the Simulator executes one of the transitions, if there is one whose precondition holds; if not, it halts. There are no guarantees about randomness or completeness; in fact, the Simulator may choose the same transition at every step, if its precondition continues to be true.

If any of the transitions have parameters, however, or if the user wants to specify the order of the transitions, then the IOA program should include a `schedule` block, as described in Section 2.5. The Simulator follows this schedule, checking each transition's preconditions and then executing its effects. If the IOA program includes invariants, then the Simulator checks that the invariants hold after each transition.

In addition to the nondeterminism of scheduling, IOA allows explicit nondeterminism in the form of `choose` statements. The user must tell the Simulator how to resolve these choices; Section 2.6 describes how to do so.

2.4.2 Supported IOA sorts

The Simulator currently has implementations for several built-in primitive IOA types (Boolean, Integer, Natural, Real, Char, and String), and it supports user-defined types formed from the `enumeration`, `tuple`, and `union` shorthands, and those formed from the `Array` (for one-dimensional arrays), `Seq` (sequence), `Set`, `Mset` (multiset), and `Map` constructors. These types, shorthands, and constructors are described in the IOA Manual [6].

There is currently no implementation for the two-dimensional use of `Array`. The front end will accept it, but the Simulator cannot yet run it.

Additionally, LSL files and Java implementations are available for the parameterized sorts `Stack`, `Tree`, and `PQ` (priority queue). These sorts are, unfortunately, not yet included in the IOA Manual.

2.4.3 Adding new data types

It is possible to add new data types to the Simulator (and, simultaneously, the Code Generator). Doing so requires writing an LSL specification and a Java implementation of each data type. The writer must ensure that the Java class implements the LSL specification; the Simulator makes no attempt to check. Instructions appear in Michael J. Tsai's paper on the design and implementation of the IOA shared data types [12].

2.4.4 `NonDet` functions

Section 2.6 provides an example of using the `randomInt` function. `NonDet.lsl` includes two other functions, and the Simulator handles them both: `randomNat` and `randomBool`. The `randomNat` function, like `randomInt`, takes 2 parameters, and returns a number between them (inclusive); `randomBool` takes no parameters.

2.4.5 Shortcut operators

The Simulator shortcuts the $\wedge$, $\vee$, and $\Rightarrow$ operators, i.e., it evaluates the first argument before deciding whether to evaluate the second. This enables handling of statements such as `size(mySeq)>0` $\wedge$ `head(mySeq)=`x, where x is some value. If the order of the arguments to the $\wedge$ were reversed, and the size of `mySeq` were zero, the Simulator would report an error due to the attempt to take the head of an empty sequence.

2.4.6 Existential and universal quantifiers

The Simulator has the ability to handle existential and universal quantifiers only when the quantified variable is an enumeration. This feature should be extended in the future to cover other, more useful cases.

To work around this limitation, an existential quantifier can often be replaced by a `choose` statement with a suitable `where` clause [3, page 39].

2.4.7 `for` loops

The Simulator does not handle `for` loops in any automaton or schedule. 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 include a mechanism for declaring a variable, and so the variable `i` must be declared and initialized outside the loop.

2.5 Resolving scheduling nondeterminism

As mentioned above, automata are generally not deterministic. This section gives examples of resolving an automaton's scheduling nondeterminism.

Section 2.4.1 describes the conditions in which the Simulator requires a `schedule` block. The schedule tells the Simulator how to determine, at every step, which transition to execute. For a given transition definition, the schedule should specify values for its parameters (if any). The full syntax for `schedule` blocks appears in Section 4.3.

2.5.1 A simple example

The `Fibonacci` program in Listing 2-3 includes a simple example of an IOA schedule block. At the nth step, the state variable c holds the value of the nth Fibonacci number in the sequence starting with the integers given as actual parameters to `initialize`. The first line, `uses NonDet`, allows the specification to refer to one of the functions (in this case, `randomInt`) defined in `NonDet.lsl`.

Listing 2-3: `Fibonacci` with a schedule block

```
uses NonDet

automaton Fibonacci
  signature
    input initialize(x:Int, y:Int)
    internal compute
  states
    initialized:Bool := false,
    a : Int, b : Int, c : Int
  transitions
    input initialize(x, y)
      eff if (initialized = false) then
            a := y-x;
            b := x;
            c := y;
            initialized := true
          fi
    internal compute
```

```
      pre initialized = true
      eff a := b;
          b := c;
          c := a + b
  schedule                           % The schedule block determines all the
    do                               % transitions the Simulator will execute.
      fire input initialize(randomInt(0,20), 16);    % first, initialize
      while true do  fire internal compute  od       % then continue with compute
    od
```

2.6 Resolving choose nondeterminism

IOA's `choose` construct allows the user to specify that a variable takes one of several values, without specifying which one it should take in any given case. This section gives an example with `choose` and shows how to resolve it for the Simulator, using `det` blocks. The full syntax for `det` blocks appears in Section 4.3.

2.6.1 A simple example

The program in Listing 2-5 gives a simple example of the use of `det` blocks, telling the Simulator how to resolve the nondeterminism of the `choose` statements in the `Fibonacci` specification in Listing 2-4, which the Simulator cannot run.

Listing 2-4: `Fibonacci` with a choose block

```
automaton FibonacciNDChoose                          % does not run in the Simulator
  signature
    internal initialize
    internal compute
  states
    initialized:Bool := false,
    a : Int, b : Int, c : Int
  transitions
    internal initialize
      pre initialized = false
      eff b := choose x:Int where 0 <= x /\ x <= 50;  % The Simulator doesn't know
          c := choose y:Int where 0 <= x /\ x <= 50;  % how to resolve a choose.
          a := c - b;
          initialized := true
    internal compute
      pre initialized
      eff a := b;
          b := c;
          c := a + b
```

Listing 2-5: `Fibonacci` with a choose block and a det block

```
uses NonDet

automaton Fibonacci
  signature
    internal initialize
    internal compute
  states
    initialized:Bool := false,
    a : Int, b : Int, c : Int
  transitions
    internal initialize
      pre initialized = false
      eff b := choose x:Int det do yield 5 od;         %% always use 5
          c := choose y:Int yield randomInt(0, 50);    %% choose a random Int
          a := c - b;
          initialized := true
    internal compute
      pre initialized
      eff a := b;
          b := c;
          c := a + b
```

In Listing 2-5, `initialize` has been re-written; the Simulator can resolve each `choose` statement. `yield 5` would be an acceptable synonym for `det do yield 5 od`; the surrounding `det do ... od` is required only when there are multiple statements in the program.

2.7 Paired Simulation

IOA lets a user describe a simulation relation between two automata (Section 1.2.3). The simulation relation asserts that one automaton, the *implementation automaton*, implements another, the *specification automaton* or *spec*. The user may also specify a correspondence between the transitions of the implementation automaton and that of the spec. The part of the program that describes this correspondence is called a `proof` block. It is called a `proof` block because it forms the outline of an induction proof (on the number of steps in the execution of the implementation automaton) of the simulation relation. The `proof` block itself is not technically a proof, because it does not verify that the simulation relation is valid for every reachable state.

For every transition in the spec, the `proof` block defines a corresponding sequence of statements in the implementation automaton. That program may execute, or *fire*, a transition in the implementation automaton. If the spec transition is external (input or output), then the program for the implementation must include a transition with the same trace as the spec transition. Section 4.4 gives the full syntax of `proof` blocks.

2.7.1 A simple example

Listing 2-6 specifies two automata and a simulation relation between them. The first automaton, `FiboSpec`, describes an automaton for computing a Fibonacci-style sequence, starting from some unspecified integers. Given this input, the Simulator runs `FiboImpl` and, at every step, runs the corresponding transition in `FiboSpec` and checks that the simulation relation holds.

Listing 2-6: Paired `Fibonacci`

```
automaton FiboSpec
 signature
   input compute
 states
   a:Int, b:Int, c:Int              %% no initial values specified
   so that a + b = c
 transitions
   input compute
     eff a := b;
         b := c;
         c := a + b

automaton FiboImpl                  %% identical to the Fibonacci automaton
 signature                          %% defined in Listing 2-1
   input compute
 states
   a:Int := 1,
   b:Int := 0,
   c:Int := 1
 transitions
   input compute
     eff a := b;
         b := c;
         c := a + b

forward simulation from FiboImpl to FiboSpec:
  (FiboImpl.a = FiboSpec.a /\
   FiboImpl.b = FiboSpec.b /\
   FiboImpl.c = FiboSpec.c)
  proof
    initially
      FiboSpec.a := FiboImpl.a;     %% specifies initial values for
      FiboSpec.b := FiboImpl.b;     %% the state variables of FiboSpec
      FiboSpec.c := FiboImpl.c
    for input compute do            %% "compute" in the implementation automaton
      fire input compute            %% corresponds to "compute" in the spec
    od
```

2.7.2 A simple example illustrating `using` values

Within a paired simulation, the `fire` statement has an additional capability not revealed by the previous example: it can specify a value for `choose` variables in the spec automaton, with a `using` clause. The `FiboSpec` automaton in Listing 2-7 is identical to the one in Listing 2-6, except that `compute` has been re-written to use a `choose` statement. This change requires the `proof` block to specify a value for the `choose` variable, `sum`. Listing 2-8 contains the output for `psim 4 FiboImpl FiboSpec fib-using.ioa`.

Listing 2-7: Paired `Fibonacci` with `using` clause (`fib-using.ioa`)

```
automaton FiboSpec
  signature
    internal compute
  states
    a:Int, b:Int, c:Int             %% no initial values specified
    so that a + b = c
  transitions
    internal compute
      eff a := b;
          b := c;
          c := choose sum where sum = a + b                       %% NEW CODE

automaton FiboImpl                  %% identical to the Fibonacci automaton
 signature                          %% defined in Listing 2-1
   internal compute
 states
   a:Int := 1,
   b:Int := 0,
   c:Int := 1
 transitions
   internal compute
     eff a := b;
         b := c;
         c := a + b

forward simulation from FiboImpl to FiboSpec:
  (FiboImpl.a = FiboSpec.a /\
   FiboImpl.b = FiboSpec.b /\
   FiboImpl.c = FiboSpec.c)
  proof
    initially
      FiboSpec.a := FiboImpl.a;     %% specifies initial values for
      FiboSpec.b := FiboImpl.b;     %% the state variables of FiboSpec
      FiboSpec.c := FiboImpl.c
    for internal compute do
      fire internal compute using FiboImpl.a+FiboImpl.b for sum    %% NEW CODE
    od
```

Listing 2-8: Paired `Fibonacci` output

```
[[[[ Begin initialization [[[[
%%%% Modified state variables for impl automaton:
```

```
     a --> 1
     b --> 0
     c --> 1
%%%% Modified state variables for spec automaton:
     a --> 1
     b --> 0
     c --> 1
]]]] End initialization ]]]]
[[[[ Begin step 1 [[[[
     Executed impl transition: internal compute in automaton FiboImpl
%%%% Modified state variables for impl automaton:
     a --> 0
     b --> 1
     c --> 1
     Executed spec transition: internal compute in automaton FiboSpec using 1 for sum
%%%% Modified state variables for spec automaton:
     a --> 0
     b --> 1
     c --> 1
]]]] End step 1 ]]]]
[[[[ Begin step 2 [[[[
     Executed impl transition: internal compute in automaton FiboImpl
%%%% Modified state variables for impl automaton:
     a --> 1
     b --> 1
     c --> 2
     Executed spec transition: internal compute in automaton FiboSpec using 2 for sum
%%%% Modified state variables for spec automaton:
     a --> 1
     b --> 1
     c --> 2
]]]] End step 2 ]]]]
[[[[ Begin step 3 [[[[
     Executed impl transition: internal compute in automaton FiboImpl
%%%% Modified state variables for impl automaton:
     a --> 1
     b --> 2
     c --> 3
     Executed spec transition: internal compute in automaton FiboSpec using 3 for sum
%%%% Modified state variables for spec automaton:
     a --> 1
     b --> 2
     c --> 3
]]]] End step 3 ]]]]
[[[[ Begin step 4 [[[[
     Executed impl transition: internal compute in automaton FiboImpl
%%%% Modified state variables for impl automaton:
     a --> 2
     b --> 3
     c --> 5
     Executed spec transition: internal compute in automaton FiboSpec using 5 for sum
%%%% Modified state variables for spec automaton:
     a --> 2
     b --> 3
     c --> 5
]]]] End step 4 ]]]]
```

2.8 Additional example programs

This section includes a larger IOA program and describes others that are available in the IOA Toolkit distribution.

2.8.1 Replicated data

Shien Jin Ong wrote the following version of the specifications developed in [2]. This program appears here for several reasons:

- It is a more realistic IOA example than Fibonacci is. It describes a distributed system at a level of abstraction suitable for the Simulator; the Fibonacci examples are contrived to illustrate certain IOA and Simulator features.
- It is one of the largest examples that the IOA group has run in the Simulator.
- It contains examples of IOA enumerations and unions.

The program in Listing 2-9 contains five sections:

1. Declarations of `Message` and `Invocation`. An `Invocation` is either a value, which is a `Nat` to be written to memory, or a `Message`: "read," "OK," or "nil."
2. `CentralMem`, an automaton holding one value (a natural number) in memory. Each action is parameterized by a `Nat` representing one of five clients using the memory. The arrays `req` and `resp` hold requests and responses for each client.
3. `StrongCache`. It represents the same function, performed by a central memory (represented by the state variable `mem`) and five caches, one for each client.
4. A forward simulation relation from `StrongCache` to `CentralMem`.
5. A forward simulation relation from `CentralMem` to `StrongCache`.

Ong has also written formal hand proofs of the two simulation relations [10].

Listing 2-9: Strong cache and central memory

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Number of nodes limited to 5.  %
% Write/Memory values = [1, 100] %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%--- 1. type declarations ---%
uses NonDet

type Message = enumeration of read, OK, nil
type Invocation = union of val: Nat, msg: Message

%--- 2. CentralMem ---%
automaton CentralMem
  signature
    input invoke(v: Invocation, n: Nat)       % invocation from client n
    output respond(r: Invocation, n: Nat)     % response to client n
    internal perform(n: Nat)
  states
    mem:  Nat := 0,
    req:  Array[Nat, Invocation] := constant(msg(nil)),
    resp: Array[Nat, Invocation] := constant(msg(nil))
  transitions
    input invoke(v, n)
      eff req[n] := v
    output respond(r, n)
      pre resp[n] = r /\ resp[n] ~= msg(nil)
      eff req[n] := msg(nil);
          resp[n] := msg(nil)
    internal perform(n)
      pre req[n] ~= msg(nil) /\ resp[n] = msg(nil)
      eff if req[n] = msg(read) then resp[n] := val(mem)
          else mem := req[n].val;
               resp[n] := msg(OK)
          fi
  schedule
    states
      die: Nat,
      Node: Nat,                              % represents a client
      read_write: Bool,                       % whether to read or to write
      value: Nat                              % the value to write
    do
      while true do
        die := randomNat(1,2);
        Node := randomNat(1,5);
        read_write := randomBool;
        value := randomNat(1,100);
        if die = 1 /\ req[Node] = msg(nil) then
          if read_write then fire input invoke(msg(read), Node)
          else fire input invoke(val(value), Node)
          fi
        elseif die = 2 /\ resp[Node] ~= msg(nil) then
          fire output respond(resp[Node], Node)
        elseif req[Node] ~= msg(nil) /\ resp[Node] = msg(nil) then
          fire internal perform(Node)
        fi
      od
    od
```

```
%--- 3. StrongCache ---%
automaton StrongCache
  signature
    input invoke(v: Invocation, n: Nat)
    output respond(r: Invocation, n: Nat)
    internal perform(n: Nat), copy(n: Nat), drop(n: Nat)
  states
    mem: Nat := 0,
    cache: Array[Nat, Invocation] := constant(msg(nil)),
    req: Array[Nat, Invocation] := constant(msg(nil)),
    resp: Array[Nat, Invocation] := constant(msg(nil))
  transitions
    input invoke(v, n)
      eff req[n] := v
    output respond(r, n)
      pre resp[n] = r /\ resp[n] ~= msg(nil)
      eff req[n] := msg(nil);
          resp[n] := msg(nil)
    internal copy(n)
      eff cache[n] := val(mem)
    internal drop(n)
      eff cache[n] := msg(nil)
    internal perform(n)
      pre req[n] ~= msg(nil) /\ resp[n] = msg(nil)
          /\ (req[n] ~= msg(read) \/ cache[n] ~= msg(nil))
      eff if req[n] = msg(read) then resp[n] := cache[n]
          else mem := req[n].val;
               resp[n] := msg(OK);
               cache := constant(msg(nil))
          fi
  schedule
    states
      die: Nat,
      Node: Nat,                              % represents a client
      read_write: Bool,                       % whether to read or to write
      value: Nat                              % the value to write
    do
      while true do
         die := randomNat(1,4);
         Node := randomNat(1,5);
         read_write := randomBool;
         value := randomNat(1,100);
         if die = 1 /\ req[Node] = msg(nil) then
            if read_write then fire input invoke(msg(read), Node)
            else fire input invoke(val(value), Node)
            fi
         elseif die = 2 /\ resp[Node] ~= msg(nil) then
            fire output respond(resp[Node], Node)
         elseif req[Node] ~= msg(nil) /\ resp[Node] = msg(nil)
                /\ (req[Node] ~= msg(read) \/ cache[Node] ~= msg(nil))
            then fire internal perform(Node)
         elseif die = 3 then
            fire internal copy(Node)
         elseif die = 4 then
            fire internal drop(Node)
         fi
      od
    od
```

```
%--- 4. simulation relation, with proof block ---%
forward simulation from StrongCache to CentralMem:
   StrongCache.mem  = CentralMem.mem
/\ StrongCache.resp = CentralMem.resp
/\ StrongCache.req  = CentralMem.req
/\ (StrongCache.cache[1] = msg(nil) \/ StrongCache.cache[1].val = CentralMem.mem)
/\ (StrongCache.cache[2] = msg(nil) \/ StrongCache.cache[2].val = CentralMem.mem)
/\ (StrongCache.cache[3] = msg(nil) \/ StrongCache.cache[3].val = CentralMem.mem)
/\ (StrongCache.cache[4] = msg(nil) \/ StrongCache.cache[4].val = CentralMem.mem)
/\ (StrongCache.cache[5] = msg(nil) \/ StrongCache.cache[5].val = CentralMem.mem)
% \A n: Nat (StrongCache.cache[n] = msg(nil) \/ StrongCache.cache[n] = CentralMem.mem)

proof
  initially
    CentralMem.mem  := 0;
    CentralMem.req  := constant(msg(nil));
    CentralMem.resp := constant(msg(nil))

  for input invoke(v: Invocation, n: Nat)
      do fire input invoke(v, n) od
  for output respond(r: Invocation, n: Nat)
      do fire output respond(r, n) od

  for internal copy(n: Nat) ignore
  for internal drop(n: Nat) ignore
  for internal perform(n: Nat)
      do fire internal perform(n) od

%--- 5. simulation relation, with proof block ---%
forward simulation from CentralMem to StrongCache:
   StrongCache.mem  = CentralMem.mem
/\ StrongCache.resp = CentralMem.resp
/\ StrongCache.req  = CentralMem.req
/\ (StrongCache.cache[1] = msg(nil) \/ StrongCache.cache[1].val = CentralMem.mem)
/\ (StrongCache.cache[2] = msg(nil) \/ StrongCache.cache[2].val = CentralMem.mem)
/\ (StrongCache.cache[3] = msg(nil) \/ StrongCache.cache[3].val = CentralMem.mem)
/\ (StrongCache.cache[4] = msg(nil) \/ StrongCache.cache[4].val = CentralMem.mem)
/\ (StrongCache.cache[5] = msg(nil) \/ StrongCache.cache[5].val = CentralMem.mem)
% ideally, \A n: Nat (StrongCache.cache[n] = msg(nil) \/
%                     StrongCache.cache[n] = CentralMem.mem)

proof
  initially
    StrongCache.mem  := 0;
    StrongCache.req  := constant(msg(nil));
    StrongCache.resp := constant(msg(nil));
    StrongCache.cache := constant(msg(nil))
  for input invoke(v: Invocation, n: Nat)
      do fire input invoke(v, n) od
  for output respond(r: Invocation, n: Nat)
      do fire output respond(r, n) od
  for internal perform(n: Nat)
      do fire internal copy(n);
         fire internal perform(n);
         fire internal drop(n)
      od
```

2.8.2 Pointers to other examples

The IOA Toolkit distribution contains a `Test` directory, which includes many IOA examples. Many of them are simple, designed to test implementations of specific IOA sorts; these examples are usually named for the sorts in question. Other tests, such as Toh Ne Win's `Banking01` and `Banking02` and Ramírez' `MutEx01`, are better examples of IOA programs.

`Char01`, `Int01`, `Nat01`, `Real01`, and `String01` test operations on characters, integers, natural numbers, real numbers, and strings, respectively.

`Array01` tests one- and two-dimensional arrays. `Array04` includes an array indexed by an enumeration. `Array02`, `Array03`, and `Array05` also test the various operations on arrays. `Map01` tests operations on a map. `Mset01` and `Mset02` test operations on multisets. `Set01` tests operations on a set of integers. `Set02` tests operations on a set of tuples; it is intended to check whether the Simulator is producing output in a format useful to Daikon. `Seq01` tests operations on a sequence.

`Enum01`, `Tuple01`, `Tuple02`, and `Union01` are simple tests of enumerations, tuples, and unions. `Tuple03` and `Union02` test tuples and unions that use a parameterized sort (`Set[Int]`, a set of integers).

`ShortcutAnd02`, `ShortcutImplies02`, and `ShortcutOr02` test that $\wedge$, $\Rightarrow$, and $\vee$ shortcut in the Simulator. `PQ01`, `PQ02`, `Stack01`, and `Tree01` test the newest sorts available for the Simulator and CodeGen: a priority queue, a stack, and a tree.

`http://theory.lcs.mit.edu/tds/papers/Dean/thesis.html` contains links to all IOA files used in this thesis.

Chapter 3

An IOA interface for Daikon

This chapter describes the connection between the IOA Simulator and Daikon, an invariant-discovery tool [4]. This connection has two main uses:

- Daikon can detect invariants that turn out to be verifiable; these invariants can lead to proof of an algorithm's correctness.
- Daikon can find invariants that the programmer knows should not always be true; these invariants can point to holes in a schedule block's coverage of possible cases. Both an automaton's schedule block and a traditional program's test suite determine executions and check that their targets work as expected.

This chapter gives a brief description of Daikon, describes the IOA interface to Daikon, and then gives three examples of that interface in action.

3.1 Daikon background

Daikon is a tool for detecting invariants, properties that always hold at given points in a program. It is a dynamic tool, extracting information from executions of a program rather than relying only on static information about the code itself.

Like the Simulator, Daikon is a heuristic tool rather than a rigorous proof generator: it returns invariants that are true for every execution it has ever seen, but there is no guarantee that these invariants hold in all possible executions. Daikon's purpose, however, is different from the Simulator's: it seeks to detect potential invariants by inspecting given executions of code. In contrast, the Simulator creates a possible execution, and then checks any invariants provided by the programmer.

Daikon is meant to be used on already-written code, to reveal invariants that may be vital to the code's correctness, and thus useful for maintainers to know. In the Fibonacci automaton, for example, it detects the following invariant: `Fibonacci.c == Fibonacci.a + Fibonacci.b`. In a Banking automaton developed by Toh Ne Win, Daikon detects the following invariant: `size(Bank.actives) = size(Bank.bals) + size(Bank.pending_ops)`, which indicates that every active request is either a request for a balance (a read) or a request to perform an operation (a write).

In the IOA paradigm, a programmer first models an algorithm with IOA, then verifies its correctness, and finally creates a distributed implementation,[1] possibly using IOA's CodeGen tool. For IOA, Daikon can be used to detect invariants that might be helpful in proving an automaton's correctness.

3.1.1 Daikon's input format

Daikon examines information about given executions of a program. As its input, it requires two kinds of information: declarations and data traces. The two kinds of information are usually recorded in two separate files, but this is not required. The Simulator always records two separate files.

Declaration files contain lists of program points considered interesting to the user, with a list of variables in scope at each program point. Data trace files contain information about runtime values of variables: 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.

Daikon recognizes some program point names as having special meaning. Program points `Foo:::ENTER` and `Foo:::EXIT` are interpreted as entry and exit points of a method (or function, or transition, depending on the programming language) named

[1] An IOA program itself is arguably an implementation, if it runs in the Simulator, but it is not a distributed implementation.

`Foo`; Daikon attempts to find invariants relating the pre- and post-states of that method.

3.2 Motivation for using the Simulator for Daikon

Daikon already has a Java front end, and CodeGen will eventually produce output in Java. Thus the IOA→Java→Daikon route could suffice as an IOA-Daikon interface. This route has an obvious advantage: completion of the IOA code generator implies completion of an interface to Daikon, with no extra work. The disadvantage of this approach, however, lies in the addition of an extra layer. Daikon (and the programmer) would be reasoning about the intermediate Java code, rather than the original IOA program. Additionally, the Java code would include low-level networking and I/O details, which are presumably not the "interesting" part of the program. This inclusion would increase the size of the space searched by Daikon, with little benefit to the programmer.

For those reasons, it is cleaner for the Simulator to produce the input for Daikon to examine. In addition to the gains in elegance, using the Simulator has a practical advantage: because of the simplicity of the required modifications, the Daikon-output prototype did not take long to produce, in contrast to the more complex Java code generator, which is not yet available.

3.3 Implementation of the Daikon interface

The implementation began as a simple prototype, which declared a program point for the end of every transition, and declared all of the automaton's state variables to be in scope at every program point. Even at that stage, it was able to find several invariants in the Fibonacci example (discussed below).

Since then, Toh Ne Win has added the following capabilities:

- The trace file now declares a program point for the entry and the exit of every transition; this gives Daikon information about how a transition's pre-state relates to its post-state, enabling it to detect relations that always hold between the two, as mentioned in Section 3.1.1.

- The trace file also declares an `automatonName:::OBJECT` program point, permitting Daikon to detect invariants that hold at all times, not just at certain points. This point is called "object" by analogy to object invariants, which are also known as representation invariants.

- The Simulator now uses a separate Listener for Daikon output, improving the modularity of the program. This detail is not interesting to the user, but it should be helpful to future maintainers. For information on the Simulator's use of Events and Listeners, see Section 6.6 of Ramírez' thesis [11].

The output given in this thesis is generated by this new version of the IOA-Daikon interface.

3.4 Fibonacci example

The `Fibonacci` automaton from Listing 2-1 is reproduced for convenience here in Listing 3-1. This automaton is simple (no schedule block is required) and has an easily-understood invariant: a+b=c. For these reasons, it is an excellent first example, and it was a good first test case for the IOA connection to Daikon. The declarations and data trace files appear in Appendix B.1.

Listing 3-1: `Fibonacci.ioa`

```
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 A of Fibonacci:
  a + b = c
```

Even on this simple example, Daikon produced invariants that the programmer hadn't thought to write down, such as `a <= c` and `b <= c`. Indeed, this points to one of the strengths of Daikon: its ability to point out invariants that the programmer thought were obvious, but that a later maintainer might not know were important. Daikon's output for the Fibonacci example follows, with comments (indicated by "%") added. This output was produced by running `java daikon.Daikon Fibonacci.decls Fibonacci.dtrace`, where the .decls and .dtrace files were generated by a Simulator run of ten steps (as given in Appendix B.1); the output does not change if the Daikon input files are instead generated by Simulator runs of 20, 30, or 40 steps.

Listing 3-2: Daikon output for `Fibonacci`

```
Fibonacci:::OBJECT
Fibonacci.a <= Fibonacci.c
Fibonacci.b <= Fibonacci.c
Fibonacci.c == Fibonacci.a + Fibonacci.b
===========================================================================
Fibonacci.compute():::ENTER
===========================================================================
Fibonacci.compute():::EXIT
Fibonacci.a == orig(Fibonacci.b)
Fibonacci.b == orig(Fibonacci.c)
Fibonacci.a <= Fibonacci.b
Fibonacci.a < Fibonacci.c
Fibonacci.b >= orig(Fibonacci.a)
Fibonacci.c >= orig(Fibonacci.a)
orig(Fibonacci.a) == - Fibonacci.a + Fibonacci.b
orig(Fibonacci.a) == - 2 * Fibonacci.a + Fibonacci.c
orig(Fibonacci.a) == 2 * Fibonacci.b - Fibonacci.c
```

3.5 Parameterized Fibonacci example

The IOA program in Listing 3-3 runs `Fibonacci` on two integers `x` and `y` determined by the automaton's schedule block: $0 \leq \texttt{x} \leq 30$ and $-20 \leq \texttt{y} \leq 20$. As in Section 3.4, a+b=c is an expected invariant. Since the state variables are initialized to different values in different test runs, some of the invariants in Listing 3-2, such as `b < c`, are expected to disappear.

Listing 3-3: `Fibonacci` with unspecified parameters

```
uses NonDet

automaton Fibonacci
  signature
    input initialize(x:Int, y:Int)
    internal compute
  states
    initialized:Bool := false,
    a : Int, b : Int, c : Int
    so that a + b = c
  transitions
    input initialize(x, y)
      eff if (initialized = false) then
            a := y - x;
            b := x;
            c := y;
            initialized := true
          fi
    internal compute
      pre initialized = true
      eff a := b; b := c; c := a + b
  schedule
    do
      fire input initialize(randomInt(0,30), randomInt(-20, 20));
      while true do  fire internal compute  od
    od
```

This program was run ten times, producing ten .dtrace files. Each run consisted of 35 steps: the data trace file for the nth run was generated by `sim -daikon -ignoreFirst -odtrace fibn.dtrace 35 fib-param.ioa`. `-ignoreFirst` means that no values are printed for the `OBJECT` program point until the first transition: in this case, `initialize`. This flag was used because the Simulator initializes the state variables `a`, `b`, and `c` to values that do not satisfy the desired invariant: `a+b=c`. It would be nice to have the Simulator initialize `a` and `b` to unspecified values, and then initialize `c` to `a+b`, but IOA does not allow a given state variable's initialization value to refer to values of other state variables. In addition, the Simulator currently ignores the `so that a+b=c` clause; this is an oversight in the implementation of the Simulator, and Section 7.2.3 discusses possible solutions.

The Daikon results were generated by `java daikon.Daikon fib-paired.decls fib*.dtrace`. These results appear in Listing 3-4.

Listing 3-4: Daikon output for parameterized `Fibonacci`

```
Fibonacci:::OBJECT
Fibonacci.initialized == true
Fibonacci.c == Fibonacci.a + Fibonacci.b
===========================================================================
```

```
Fibonacci.compute:::ENTER
Fibonacci.initialized == true
Fibonacci.c == Fibonacci.a + Fibonacci.b
===========================================================================
Fibonacci.compute:::EXIT
Fibonacci.initialized == orig(Fibonacci.initialized)
Fibonacci.a == orig(Fibonacci.b)
Fibonacci.b == orig(Fibonacci.c)
Fibonacci.c == Fibonacci.a + Fibonacci.b
orig(Fibonacci.a) == - Fibonacci.a + Fibonacci.b
orig(Fibonacci.a) == - 2 * Fibonacci.a + Fibonacci.c
orig(Fibonacci.a) == 2 * Fibonacci.b - Fibonacci.c
===========================================================================
Fibonacci.initialize:::ENTER
Fibonacci.a == Fibonacci.b == Fibonacci.c
Fibonacci.initialized == false
Fibonacci.a == 87
===========================================================================
Fibonacci.initialize:::EXIT
orig(Fibonacci.a) == orig(Fibonacci.b) == orig(Fibonacci.c)
Fibonacci.initialized == true
Fibonacci.a <= Fibonacci.c
Fibonacci.c == Fibonacci.a + Fibonacci.b
```

Sometimes Daikon produces invariants that the writer of the automaton would expect not to be true all the time. For example, `Fibonacci.a <= Fibonacci.c` at the exit point of `initialize` indicates that `b` was always initialized to a positive value; this could be construed as an omission in the test suite (schedule). In this case, the source of the omission is fairly clear from inspection of the schedule block, but one could imagine a more complicated schedule block. Similarly, `a == 87` at the entry point of `initialize` is an artifact of the ADT implementation of `Int`, `ioa.runtime.IntSort`, which initializes all integers to 87 by default. 87 is a rather non-traditional initialization value, but its use is not a bad thing. A reader who sees `a == 87` will probably not simply accept it as an invariant, without thinking, but will instead notice that the invariant is an artifact of the implementation.

3.6 Hanoi example

Listing 3-5 contains the `Hanoi` automaton, an IOA implementation of the Towers of Hanoi problem for three poles and five discs. The automaton's schedule block solves the problem, moving the discs from stack A to stack C in 31 steps. This Hanoi program illustrates one of the limitations of IOA: the lack of a convenient way to define functions. It is possible to define functions, but it requires writing both LSL and Java. The schedule includes substantial repetition, and the `putOn` transitions are all quite similar.

Listing 3-5: `Hanoi.ioa`

```
uses Stack(Int)

automaton Hanoi
  signature
    internal putOnA(d: Int), putOnB(d: Int), putOnC(d: Int)
    internal start
  states
    stackA : Stack[Int] := empty,
    stackB : Stack[Int] := empty,
    stackC : Stack[Int] := empty
  transitions
    internal start                                 % initialize: 5 discs on stack A
      pre
        stackA = empty /\ stackB = empty /\ stackC = empty
      eff
        stackA := push(1, push(2, push (3, push (4, push(5, empty)))))
    internal putOnA(d: Int)                        % put disc d onto stack A
      pre (stackA = empty \/ top(stackA) > d) /\
          ((stackB ~= empty /\ d = top(stackB)) \/ (stackC ~= empty /\ d = top(stackC)))
      eff
        stackA := push(d, stackA);
        if (stackB ~= empty /\ d = top(stackB)) then stackB := pop(stackB)
        else stackC := pop(stackC)
        fi
    internal putOnB(d: Int)                        % put disc d onto stack B
      pre (stackB = empty \/ top(stackB) > d) /\
          ((stackA ~= empty /\ d = top(stackA)) \/ (stackC ~= empty /\ d = top(stackC)))
      eff
        stackB := push(d, stackB);
        if (stackA ~= empty /\ d = top(stackA)) then stackA := pop(stackA)
        else stackC := pop(stackC)
        fi
    internal putOnC(d: Int)                        % put disc d onto stack C
      pre (stackC = empty \/ top(stackC) > d) /\
          ((stackA ~= empty /\ d = top(stackA)) \/ (stackB ~= empty /\ d = top(stackB)))
      eff
        stackC := push(d, stackC);
        if (stackA ~= empty /\ d = top(stackA)) then stackA := pop(stackA)
        else stackB := pop(stackB)
        fi

schedule                                           % move 5 discs to stack C
  do
    fire internal start;
    fire internal putOnC(1); fire internal putOnB(2); fire internal putOnB(1);
    fire internal putOnC(3);
    fire internal putOnA(1); fire internal putOnC(2); fire internal putOnC(1);
    fire internal putOnB(4);
    fire internal putOnB(1); fire internal putOnA(2); fire internal putOnA(1);
    fire internal putOnB(3);
    fire internal putOnC(1); fire internal putOnB(2); fire internal putOnB(1);
    fire internal putOnC(5);
    fire internal putOnA(1); fire internal putOnC(2); fire internal putOnC(1);
```

```
    fire internal putOnA(3);
    fire internal putOnB(1); fire internal putOnA(2); fire internal putOnA(1);
    fire internal putOnC(4);
    fire internal putOnC(1); fire internal putOnB(2); fire internal putOnB(1);
    fire internal putOnC(3);
    fire internal putOnA(1); fire internal putOnC(2); fire internal putOnC(1)
  od

invariant Size of Hanoi:            % should be false until "start" transition
  size(stackA) + size(stackB) + size(stackC) = 5
```

The command `sim -daikon 100 Hanoi.ioa` produces .decls and .dtrace files. `java daikon.Daikon -o hanoi.inv Hanoi.decls Hanoi.dtrace` finds 100 potential invariants and stores its results in a file called `hanoi.inv`. The command `java daikon.PrintInvariants --suppress_post hanoi.inv` turns on Daikon's option to "suppress display of obvious postconditions on prestate," resulting in the 83 invariants shown in the Appendix, in Listing B-3.

It is rather difficult to wade through 83 invariants, many of which are uninteresting or not true in general. For example, the invariants at the exit point of `putOnA` include `size(stackC) == 1 (mod 2)`, which is a result of the fact that the schedule block plays the game well and that the game has an odd number of discs, moving from stack A to stack C. The size of the list makes it difficult to notice interesting invariants, such as the `subsequence` relations at the exit point of `putOnA`.

The list becomes much shorter when it is the result of more than one test run. Listing 3-6 is the result of running and analyzing six different programs, moving the five discs from A to B, A to C, B to A, B to C, C to A, and C to B. With Daikon's "suppress_post" option, 40 invariants are printed. Some of these invariants result from the fact that the test cases all use exactly five discs; it would be possible to add cases with different numbers of discs.

Listing 3-6: Daikon output for `Hanoi`

```
% java daikon.Daikon -o hanoi.inv Hanoi.dtrace Hanoi.*.decls
% java daikon.PrintInvariants --suppress_post hanoi.inv
===========================================================================
Hanoi:::OBJECT
Hanoi.stackA[] contains no duplicates
Hanoi.stackA[] elements != null
Hanoi.stackB[] contains no duplicates
Hanoi.stackB[] elements != null
Hanoi.stackC[] contains no duplicates
Hanoi.stackC[] elements != null
```

```
size(Hanoi.stackC[]) == - size(Hanoi.stackA[]) - size(Hanoi.stackB[]) + 5
===========================================================================
Hanoi.putOnA():::ENTER
===========================================================================
Hanoi.putOnA():::EXIT
size(Hanoi.stackA[])-1 == orig(size(Hanoi.stackA[]))
size(Hanoi.stackA[]) >= 1
orig(Hanoi.stackA[]) is a subsequence of Hanoi.stackA[]
Hanoi.stackB[] is a subsequence of orig(Hanoi.stackB[])
Hanoi.stackC[] is a subsequence of orig(Hanoi.stackC[])
size(Hanoi.stackB[]) <= orig(size(Hanoi.stackB[]))
size(Hanoi.stackC[]) <= orig(size(Hanoi.stackC[]))
orig(size(Hanoi.stackC[])) == - size(Hanoi.stackA[]) - orig(size(Hanoi.stackB[])) + 6
===========================================================================
Hanoi.putOnB():::ENTER
===========================================================================
Hanoi.putOnB():::EXIT
size(Hanoi.stackB[])-1 == orig(size(Hanoi.stackB[]))
size(Hanoi.stackB[]) >= 1
Hanoi.stackA[] is a subsequence of orig(Hanoi.stackA[])
orig(Hanoi.stackB[]) is a subsequence of Hanoi.stackB[]
Hanoi.stackC[] is a subsequence of orig(Hanoi.stackC[])
size(Hanoi.stackA[]) <= orig(size(Hanoi.stackA[]))
size(Hanoi.stackC[]) <= orig(size(Hanoi.stackC[]))
orig(size(Hanoi.stackC[])) == - size(Hanoi.stackB[]) - orig(size(Hanoi.stackA[])) + 6
===========================================================================
Hanoi.putOnC():::ENTER
===========================================================================
Hanoi.putOnC():::EXIT
size(Hanoi.stackC[])-1 == orig(size(Hanoi.stackC[]))
size(Hanoi.stackC[]) >= 1
Hanoi.stackA[] is a subsequence of orig(Hanoi.stackA[])
Hanoi.stackB[] is a subsequence of orig(Hanoi.stackB[])
orig(Hanoi.stackC[]) is a subsequence of Hanoi.stackC[]
size(Hanoi.stackA[]) <= orig(size(Hanoi.stackA[]))
size(Hanoi.stackB[]) <= orig(size(Hanoi.stackB[]))
orig(size(Hanoi.stackB[])) == - size(Hanoi.stackC[]) - orig(size(Hanoi.stackA[])) + 6
===========================================================================
Hanoi.start():::ENTER
size(Hanoi.stackA[]) == size(Hanoi.stackB[]) == size(Hanoi.stackC[])
Hanoi.stackA[] == []
Hanoi.stackB[] == []
Hanoi.stackC[] == []
size(Hanoi.stackA[]) == 0
===========================================================================
Hanoi.start():::EXIT
orig(size(Hanoi.stackA[])) == orig(size(Hanoi.stackB[])) == orig(size(Hanoi.stackC[]))
size(Hanoi.stackA[]) one of { 0, 5 }
size(Hanoi.stackB[]) one of { 0, 5 }
size(Hanoi.stackC[]) one of { 0, 5 }
```

Chapter 4

IOA Syntax Extensions

The IOA Simulator, as described in Ramírez' thesis [11], worked with an old version of the IOA front end. His thesis gives an excellent description of the changes he made to the grammar of that front end. The new front end, however, has a slightly different grammar, and this chapter describes the Simulator-related changes to it. Additionally, it describes the semantic constraints on these extensions to the IOA language.

In this chapter, Sections 4.1 and 4.2 describe syntax extensions that were incorporated into the new front end's grammar with no changes from Ramírez' version. They appear here for two reasons: to clarify the semantic constraints, or lack thereof, on labels for invariants and transition cases, and to create a self-contained description of this part of the language. Sections 4.3 and 4.4 describe syntax extensions that are slightly different from Ramírez' version.

4.1 Labeling invariants

At every step of execution, the Simulator checks any invariants associated with each automaton. When an invariant fails, it issues an error message. If there is more than one invariant, however, an "invariant failed" message is not helpful unless there is some way to specify which invariant has failed.

To solve this problem, Ramírez introduced a way for the user to name invariants.

For the new IOA front end, this change is identical to the one described in Section 5.3 of Ramírez' thesis:

```
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 approach is not terribly useful, but it is perfectly legal.

4.2 Labeling transition definitions

It is possible for a user to define two transition definitions with identical names and parameters. Ramírez introduced the `case` keyword to allow the user to label such transitions and to refer to them unambiguously in a schedule block.

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

Listing 4-1: `BoolToggler.ioa`, legal

```
automaton BoolToggler
  signature
    internal action1
  states
    alpha: Bool := false
  transitions
    internal action1 case X
      pre alpha
      eff alpha := false
    internal action1 case X
      pre ~alpha
      eff alpha := true

%% no schedule block,
%% and no error

%% (also runs in the Simulator)
```

Listing 4-2: `BoolToggler.ioa`, illegal

```
automaton BoolToggler
  signature
    internal action1
  states
    alpha: Bool := false
  transitions
    internal action1 case X
      pre alpha
      eff alpha := false
    internal action1 case X
      pre ~alpha
      eff alpha := true

schedule do
  while true do
    fire internal action1 case X
  od
od
```

This change is identical to the one described in Ramírez' thesis:

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

In the `BoolToggler` automaton of Listing 4-2, the error occurs when the checker tries to match the transition in the `fire` statement with one of those defined in the automaton. Thus the front end prevents the introduction of ambiguity in a block whose purpose is to resolve nondeterminism.

4.3 Resolving nondeterminism

Consider the automaton defined in Listing 4-3, which illustrates both `choose` and scheduling nondeterminism. At a given step, the Simulator must decide between `action1` and `action2` (with an appropriate value for n). In addition, if `action1` is performed, then the Simulator must choose a value for x.

Listing 4-3: `Chooser.ioa`, nondeterministic

```
automaton Chooser
  signature
    output action1, action2(n:Int)
  states
    chosen: Int % initially arbitrary
  transitions
    output action1
      eff chosen := choose x where 1 <= x /\ x <= 30
    output action2(n)
      pre n = chosen
```

Now consider the IOA program in Listing 4-4, the `ResolvedChooser` automaton with a schedule block, which resolves the nondeterminism of `Chooser`.

Listing 4-4: `ResolvedChooser.ioa`, deterministic

```
automaton ResolvedChooser
  signature
    output action1, action2(n:Int)
  states
    chosen: Int := 22
  transitions
    output action1
      eff chosen := choose x where 1 <= x /\ x <= 30
```

```
                    det do
                          yield 1; yield 2; yield 3
                        od
   output action2(n)
     pre n = chosen
 schedule do
   while true do
     fire output action1;
     fire output action2(chosen)
   od
 od
```

This example, taken from Ramírez' thesis [11], illustrates his approach to resolving nondeterminism. The Chooser automaton is pure IOA, simply defining two kinds of transitions with their preconditions and effects. The ResolvedChooser contains the same code with `det` and `schedule` blocks added to produce a deterministic schedule for execution. Incidentally, the Simulator also supports a method for picking a random integer within a given range. In the example above, `yield 1; yield 2; yield 3` could be replaced by `yield randomInt(1,30)`. See the User's Guide (Chapter 2 of this thesis) for more information on its usage.

Ramírez describes these IOA language extensions in Section 5.4.2 of his thesis [11]. The following sections give the current specifications for the Simulator extensions to the grammar.

4.3.1 Scheduling transitions

These rules define the syntax of `schedule` blocks. The grammar and the implementation are both quite similar, but not identical, to Ramírez'.

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. (It was not checked in Ramírez' version of the front end; this addition should be considered a bug-fix. Discussion with Ramírez confirms that the omission was unintentional.) The other change, which is discussed below, is a change in the implementation rather than the resulting language.

4.3.2 Determining values within a `choose`

This extension is similar to the one made by Ramírez. The only difference is that, now, an `NDRProgramY` appears in place of an `NDRProgram`. This means that the only statements appearing in a `yield` context are those that return values; specifically, `fire` statements are disallowed. In Ramírez' implementation, this constraint was present, but it was enforced as a static semantic check.

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
```

4.4 Support for paired simulation

This extension is similar to Ramírez' version. Section 2.7 of this thesis contains examples of using the `proof` block to specify relations between two automata.

As in Ramírez' version, a proof is allowed only in a forward simulation context. In his version, this constraint was enforced during static semantic checking; in the new front end, however, it is part of the grammar itself.

Another, more substantial change is that an assignment in a `simProofInit` block may now assign to an `lvalue` rather than only a `variable`. As in the old version, the left-hand side of the assignment must refer to a state variable of the specification automaton, of course. And, as before, the user assumes the burden of ensuring that the `initially` assignments result in a reachable state of the specification automaton.

Original:

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

Modified:

```
simulation      ::=  'forward' 'simulation' 'from' automatonName
                     'to' automatonName ':' predicate simProof?
                   | 'backward' 'simulation' 'from' automatonName
                     'to' automatonName ':' predicate
simProof        ::=  'proof' states? simProofInit? simProofEntry+
simProofInit    ::=  'initially' (lvalue ':=' term);+
simProofEntry   ::=  'for' actionType actionName
                     actionFormals? transCase?
                     (('do' proofProgram 'od') | 'ignore')
proofProgram    ::=  proofStatement;+
proofStatement  ::=  assignment
                   | proofConditional
                   | proofWhile
                   | proofFire
proofConditional ::= 'if' predicate 'then' proofProgram
                     ('elseif' predicate 'then' proofProgram)*
                     ('else' proofProgram)? 'fi'
proofWhile      ::=  'while' predicate 'do' proofProgram 'od'
proofFire       ::=  'fire' actionType actionName
                     actionActuals? transCase?
                     ('using' ( term 'for' variable ),+)?
```

Chapter 5

The intermediate language (IL)

IOA back-end tools (such as the Simulator) rely on an intermediate language (IL) produced by the front end. This language, based on the S-expressions of Lisp, is described only briefly in Ramírez' thesis [11]. It is based on Anna Chefter's design [3], but has changed significantly since then. Once the design of the Composer (and the related IL) has been finalized, the IOA manual should be updated to include a complete description of the IL.

5.1 Simulator-related IL extensions

The following extensions to the IL reflect the IOA language extensions described in Chapter 4.

5.1.1 `det` blocks

```
<yieldprogram>     ::= (det <yieldstatement>*)
<yieldstatement>   ::= <conditional> | <ndrwhile> | <ndryield>
<ndryield>         ::= (yield <term>)
```

Any `conditional` or `ndrwhile` within a `yieldprogram` should represent a tree whose leaves are all `ndryield`; assignments and `fire`-type statements are not allowed. (This is a reflection of the similar constraint on the IOA language, as described in Section 4.3.)

5.1.2 `schedule` blocks

```
<schedule>          ::= (schedule (states <state>*) <ndrprogram>)
<ndrprogram>        ::= ({<assignment> | <conditional> |
                          <ndrfire> | <ndrwhile>}*)

<ndrfire>           ::= (fire {<transitionId> <actionActuals>?}?)
<ndrwhile>          ::= (while <predicate> <program>)
```

The transition ID for an `ndrfire` must match the ID of a transition definition with compatible parameters.

5.1.3 Paired simulation

```
<simulation>        ::= (sim forward <automatonName> <automatonName>
                             <predicate> <proof>?)
                      | (sim backward <automatonName> <automatonName>
                             <predicate>)

<simfire>           ::= (sim_fire <transitionId> <actionActuals>?
                                  (using (<variable> <term>)+)?)

<proof>             ::= (proof (states <state>*) ((<term> <value>)*)
                               (<simcorresp>*))
<simcorresp>        ::= (sim_entry <transitionId> (formals <formal>*)
                                   <program>)
```

The `term`, `value` pairs in the `proof` definition correspond to the pairs appearing in the `initially` block of the proof.

5.1.4 Invariant names

```
<invariant> ::= (invariant <invariantName> <automatonName> <predicate>)
```

5.2 Simulator IL Requirements

State variable names should appear in the IL in the same order as in their declaration in the IOA program. This constraint allows the Simulator to display them in the order the user expects.

As described in the user's guide (Section 2.4.5 of this thesis), the simulator short-cuts $\wedge$ and $\vee$, so it is important that the arguments to these operators appear (in the IL) in the same order as they do in the IOA. For the Simulator's purposes, these operators are not commutative.

5.3 Enumerations, tuples, and unions

These notes, though far from complete, form the beginning of a sections of a general IL specification. Developers of IOA back-end tools would appreciate such a document.

```
<enumeration>      ::= (enum <sortId> <succOp> <enumItem>+)
<enumItem>         ::= (<enumItemName> <enumItemOp>)
<tuple>            ::= (tuple <sortId> <opId> <field>+)
<field>            ::= (<id> <sortId> <opId> <opId>)
<union>            ::= (union <sortId> <sortId> <opId> <field>+)
```

The first `sortId` is the sort of the union. A union named "foo" must have an associated enumeration named "foo_tag" whose items are the names of the union's tags; the second sortId is the sort of that enumeration.

For example, the IOA code `type Request = union of a:Bool, b:Nat` results in the IL code in Listing 5-1.

Listing 5-1: `union.il`

```
(ioa
 ((sorts
   (s0 "Bool" () (scope 0))
   (s1 "type" () (scope 0))
   (s20 "Request" () (scope 2))
   (s21 "Request_tag" () (scope 2)))
  (ops
   ...
   (op313 (select "a") ((s20) s0) (scope 2))
   (op314 (id "a") ((s0) s20) (scope 2))
   (op315 (select "b") ((s20) s0) (scope 2))
   (op316 (id "b") ((s0) s20) (scope 2))
   ...
   (op321 (id "a") (() s21) (scope 2))
   (op322 (id "b") (() s21) (scope 2))
   (op323 (id "succ") ((s21) s21) (scope 2))
   (op324 (id "tag") ((s20) s21) (scope 2)))
  (vars))
 (union s20 s21 op324 ("a" s0 op314 op313) ("b" s0 op316 op315))
 (enum s21 op323 ("a" op321) ("b" op322)))
```

One recent change to the IL is that automata appear as variables, not constants. That is, for an automaton `Fibo`, the IL will declare a sort (also named `Fibo`) associated with that automaton, and there will be a variable `Fibo` (of that sort) respresenting the automaton's state variables. The sort is a tuple whose fields have the names and sorts of the state variables.

Chapter 6

Notes for the Simulator developer

This chapter gives information on modifying the Simulator. Some of this information appears in Ramírez' thesis [11] and is replicated here for completeness; the rest is either documentation of new features or new documentation of old features.

6.1 Interaction with the IL parser

As mentioned in section 1.1, the back-end IOA tools share an IL parser. Ramírez' thesis [11] includes some discussion of how the other tools can use this parser. This section covers some additional details of the interactions between it and the Simulator. His thesis and this section should, together, provide a basis for a complete document about the IL parser. (See Section 7.2.6 of this thesis for details.)

6.1.1 Adding new Simulator-aware IL elements

In the course of improving the Simulator, a developer may wish to add specialized versions of IL parse tree elements, to give those objects simulator-specific abilities. This task is not complicated, thanks to Ramírez' use of factories [5] in the design of the IL parser.

To add such an object, subclass the appropriate IL Element, and implement the appropriate Simulator interface: Evaluable, Compilable, or Assignable. Then add an appropriate method to SimILFactory, overriding the one in ioa.il.BasicILFactory. This process is described (from a less Simulator-centric point of view) in section 6.5 of Ramírez' thesis.

6.1.2 Extending the IL parser to handle new structures

Additions to IOA syntax often require additions to IL syntax. To handle a new kind of structure in the IL, the specialized ILElement (as described above) should have its own specialized parseExtension() method.

6.2 ADTs shared by the Simulator and CodeGen

The Simulator has been developed separately from CodeGen, but they share many common features, as they both solve the problem of taking an IOA program and producing Java code for execution. (The Simulator then uses that code right away; CodeGen emits it as a program to be run later.) For this reason, we now share the Java abstract data types (ADTs) representing IOA sorts.

Michael J. Tsai has written a document [12] describing these ADTs and the registry used as an interface to them. This document supersedes section 6.4 of Ramírez' thesis [11].

6.3 The IL variables representing automata

As mentioned in section 5.3, the IL now refers to an automaton using a variable rather than a constant.

The Simulator was changed quickly to adapt to this change, using a special case in the `evalVector()` method of `Simulator`: if evaluation of an object returns null, then check to see whether its name matches that of an automaton, and if so, return that automaton instead of null.

This implementation is a hack. Intuition suggests that this lookup should be done in the registry, not buried inside the `evalVector()` method. The registry was avoided, in the quick fix, because of naming issues. For example, it is perfectly legal to have an automaton named `Int`, leading to having two different sorts named `Int`: one for integers, and one for the automaton. Since the registry currently works based only on the names of sorts, and not their identifiers in the IL, a lookup for the sort named "Int" would be ambiguous in such a case.

The previous paragraph justifies the decision to "fix" the Simulator as described, but it is not a good reason for letting that fix remain in place. Indeed, the lookup should be done in the Simulator's registry; there's no reason the registry shouldn't know the names of the automata it's working for. Adding an `addAutomaton()` method to the Simulator's registry (`impl.BasicImplRegistry`) would give it the necessary information about mapping from a sort's identifier (not just its name, since that would be ambiguous) to an automaton. Then it can use that information when looking up a sort, and, where appropriate, return an `Automaton` object.

Chapter 7

Conclusion

This chapter summarizes the work of the thesis and then enumerates ideas for future work to improve the Simulator.

7.1 Summary of work

This thesis has described several improvements to the Simulator:

- providing a user's guide to the Simulator, with clear explanations of the IOA programs it can and cannot run,
- extending the Simulator to serve as an IOA interface to Daikon,
- updating the Simulator to use the new shared ADTs (working with Michael J. Tsai and Toh Ne Win), thus increasing its capabilities and improving its maintainability,
- modifying the IOA front end to include Simulator-related extensions to the IOA language, and
- beginning the work of documenting the IL.

7.2 Future work

Section 6.3 suggests a Simulator improvement that would help future maintainers but would not be user-visible. The following sections suggest additions to increase the Simulator's capabilities.

7.2.1 Resolution of nondeterminism

Ramírez suggests several extensions to make schedules smaller and less repetitious [11, Section 2.6]. In particular, he suggests providing a library of non-determinism resolution (NDR) programs for given types of variables and for given forms of `where` predicates, and providing default schedules. Default NDR programs for choosing random Booleans and for choosing random Integers and Naturals within specified ranges would be particularly helpful, since those cases are common.

Additionally, he outlines the need for articulating "simulability conditions" for IOA programs. This definition would delineate the set of automata that the Simulator could run without explicit schedule blocks. He lists four possible sets of such conditions, in increasing order of expressiveness:

1. Disallow all quantifiers, `choose` and `for` statements, and actions with formal parameters.

2. Allow actions to have formal parameters, but require them to appear as constants in each transition definition.

3. Allow formal parameters, but restrict transition definitions to the following form:

$$
\begin{array}{l}
\mathit{actionType}\ \mathit{actionName}(\mathit{var}_1 : \mathit{sort}_1, \mathit{var}_2 : \mathit{sort}_2, \ldots, \mathit{var}_n : \mathit{sort}_n) \\
\quad \textbf{pre}\ \begin{array}[t]{ll} \mathit{var}_1 & = \mathit{term}_1 \wedge \\ \mathit{var}_2 & = \mathit{term}_2 \wedge \\ & \vdots \\ \mathit{var}_n & = \mathit{term}_n \wedge \\ \mathit{restPred} & \end{array} \\
\quad \textbf{eff}\ \ldots
\end{array}
$$

where

- each $term_i$ is an IOA term which depends only on the state variables of the automaton, and not on any of the variables var_i, and
- *restPred* is an IOA predicate (without quantifiers).

This constraint has the result that at most one set of values of the variables satisfies the precondition in a given state, and that each variable's appropriate value can be found by evaluating the corresponding $term_i$.

4. Relax the restriction on transition definitions to the following form:

$$
\begin{array}{l}
\mathit{actionType}\ \mathit{actionName}(var_1 : sort_1, var_2 : sort_2, \ldots, var_n : sort_n) \\
\quad \textbf{pre}\ \ term'_1 = term_1 \wedge \\
\qquad\quad term'_2 = term_2 \wedge \\
\qquad\qquad\quad \vdots \\
\qquad\quad term'_n = term_n \wedge \\
\qquad\quad restPred \\
\quad \textbf{eff}\ \ldots
\end{array}
$$

where

- each $term'_i$ is of the form $op_i(var_i, t_{i,1}, \ldots, t_{i,r_i})$, where op_i is an operator and the $t_{i,j}$ are terms involving only the state variables of the automaton,
- each $term_i$ is an IOA term involving only the state variables of the automaton, and
- *restPred* is an IOA predicate (without quantifiers).

The third option would be a particularly approachable, yet highly useful, way to reduce the burden of writing schedule blocks for the Simulator.

7.2.2 Handling existential and universal quantifiers

Universal and existential quantifiers are quite useful in IOA programs, particularly in invariants and simulation relations. Clearly, it is not possible for the Simulator to handle arbitrary expressions involving these quantifiers — for example, $\forall$ `n:Int (pred(n))` is impossible (in general) to verify in finite time. It would be helpful, however, for the Simulator to be able to handle some useful subset of expressions involving these quantifiers.

One approach is to program the Simulator to cover the following cases:

- $\forall$ `n:Type (pred(n))`, where `Type` has a finite number of members. Of course, this raises the question of how the Simulator can know whether `Type` has this property. One option is to program the Simulator to deal with some set of IOA sorts, and hope that nobody wants to use the quantifiers with anything else. Another is to define an EnumerableADT interface (similar to the ComparableADT interface [12]) and handle only enumerable types. (This option allows the Simulator to enter infinite loops over countably infinite sets, but since the Simulator already handles `while` loops, it isn't really a new ability.)

  There are now Simulator-aware versions of ForAllTerm and ExistsTerm, and they can check that a predicate holds for all elements in an enumeration. These classes are currently hard-coded to call Sort.isEnum() to determine whether a given type is finite; this is **not** intended to be a permanent solution.

- $\forall$ `n:Type (P(n)` $\Rightarrow$ `Q(n))`, where `P(n)` fits a pattern that the Simulator knows how to handle. In particular, $x < \texttt{n} < y$ (with $x < y$) would be a useful pattern. Again, `Type` would need to be known to be enumerable.

The ideas above parallel those presented in Sections 2.6.1 and 2.6.2, respectively, of Ramírez' thesis: per-sort and per-predicate default `choose` NDR programs. Indeed, they would depend on similar, but not identical, functionality. Where the extensions here require the Simulator to be able to enumerate all elements of a set, Ramírez' proposed extensions require it to be able to pick an element of that set.

Shien Jin Ong has suggested an alternate approach: allowing the user to add `det` blocks to these predicates. A user could then write a `while` loop (or any other program) to look at "interesting" values of the quantified variable. Because one of my original goals was to reduce the amount of additional code required to make the

Simulator do "obvious" work, I do not like this suggestion as a substitute for the proposal outlined above.

A combined approach is possible: First, implement the suggestions made in the two bullet points above. Second, implement the `det` block extension to the syntax, thus allowing users to take control in the following situations:

1. The Simulator does not know that `Type` is finite; the user writes a `det` block to iterate through all possible assignments of values to the quantified variable.

2. `Type` is known by the Simulator to be finite, but is known by the user to be large (and thus time-consuming to check).

3. The predicate `P(n)` is satisfied by a finite number of elements, but it does not follow a pattern known to the Simulator.

4. The predicate `P(n)` follows one of the known patterns, but the user believes it will take too long to check. (This case is analogous to case 2.)

Some might argue that using `det` in case (2) or case (4) is not an appropriate way to treat an invariant. Perhaps so, but it's also worth noting that the Simulator cannot be used to prove that an invariant is always true. So, as long as the user realizes the implications of using the `det` block there, it is a reasonable way to save running time.

The appearance of quantifiers in preconditions (as opposed to in invariants) raises more serious concerns. The Simulator currently claims that all its executions are valid; if the user were allowed to add the proposed `det` blocks, that would no longer be the case. (This issue is relevant not only in (2) and (4), but in all the cases listed above.) For this reason, the implementor should consider disallowing `det` blocks in transition preconditions.

7.2.3 Checking `so that` clauses for state variables

As shown in Section 3.5, the Simulator does not check the `so that` clause in an automaton's `states` block. Ideally, it would be able to generate values that satisfy certain classes of `so that` clauses; in the short term, it should at least report an error if state variables are initialized to values that do not satisfy the clause associated with them.

7.2.4 The Daikon interface

The Simulator has some limitations as an interface to Daikon:

- When a new data type is introduced, how does the Simulator know whether to declare it as an array, a String, or an integer? The current method is to modify the Simulator's code. The Hanoi case, for example, required telling the Simulator that a `Stack` should be printed as an array for Daikon. This approach increases the work of adding new data types; it would be helpful for the required work to be isolated to the `ioa.runtime` and `ioa.registry` packages.

- Transition definitions sometimes have parameters; such parameters should appear as variables in scope at the `ENTER` and `EXIT` points of their transitions.

7.2.5 The IOA front end

Chapter 4 describes the current state of the front end; this section suggests future improvements.

- Once the Composer actually exists, check that the current syntax for `schedule` and `proof` blocks will work with composite automata. There may be no additional work necessary, but it is impossible to tell until the syntax and semantics of composition are finalized.

- The user has an obligation to ensure that a paired simulation's `proof` begins from a reachable state of the specification automaton. This task is simple because, before execution begins, no transitions have occurred. The front end should therefore be able to perform this check by making sure that variables initialized in an `initially` block are not also initialized at the time of declaration in the specification automaton.

7.2.6 Documentation

Chapter 2 of this thesis provides the basis for a Simulator user's guide, and Chapter 4 provides Simulator-related IOA syntax information to be added to the IOA manual. Several other parts of the IOA Toolkit, however, need substantial additional documentation work:

- IL syntax (formatting): Before this thesis, the IL syntax was documented only as Javadoc-style comments in the code, and extracted into a single file (seen in the appendix to this thesis) by a sed script. Unfortunately, the IL syntax comments use angle brackets around the names of non-terminals, and Javadoc's output is HTML. This problem could be solved with a doclet to translate angle brackets to `&langle` and `&rangle` within IL syntax comments.

- IL syntax and semantics (content): Once the Composer is complete and the IL for composed automata has been determined, its developers should document the syntax and semantics of the IL.

- The IL parser: Ramírez' thesis and the documentation produced by Javadoc are the only sources of information on how to use the IL parser. Section 6.3 of Ramírez' thesis includes a nice example of using the parser. Section 6.5 describes how to add specialized IL elements and asserts that it is possible for a back-end tool to recognize customized IL statements, but refers the reader to the IOA Toolkit documentation (which covers quite a few classes) rather than giving a description of how to do so. A unified description of the IL parser, including both how to use and how to extend it, would be useful for future back-end developers. This thesis does not include such a description because Toh Ne Win is currently re-writing parts of the IL parser.

7.2.7 IOA language wish list

Simulator users have requested the following features:

- String literals would be convenient. Sometimes enumerations provide an acceptable substitute, but sometimes they do not.

- Functions would increase readability (and ease of writing) of IOA programs. Schedules, preconditions, and effects all tend to involve too much repetition. Even the ability to write simple macros would be helpful, and seems like a more likely extension to IOA.

- One contributor to the ugliness of `schedule` blocks is that they must check a transition's precondition before firing it. The following addition would make schedules shorter and easier to read and to write: a new command, called `conditional_fire` (or some shorter name), that checks a transition's precondition and executes the transition only if its precondition is satisfied.

- Automata sometimes have transitions meant to be run only once and as the first transition that occurs; for this reason, it would be good to allow `initially` blocks of paired simulation "proofs" to contain general programs, not just assignments. (These programs should, of course, fire only internal transitions.) Conversation with Ramírez has confirmed this opinion.

Appendix A

Intermediate Language Grammar

```
<START>             ::= (ioa <decls> <spec>*)

<spec>              ::= <automatonDef> | <typedef> | <invariant> | <simulation>
                        | <trait>

<decls>             ::= (<sorts> <ops> <vars>)
<sorts>             ::= (sorts <sort>*)
<sort>              ::= (<sortId> <name> (<sortId>*) lit?)
<ops>               ::= (ops <op>*)
<op>                ::= (<opId> <opName> <signature>)
<vars>              ::= (vars <var>*)
<var>               ::= (<varId> <name> <sortId>)

<converts>          ::= (converts (<opId>+) {(exempting <term>+)}?)

<genPartyBy>        ::= (<genPartKind> <sortId> <opId>+)
<genPartKind>       ::= generated | generatedFreely | partitioned

<operator>          ::= <constant> | <numeral> | <plainOp> | <ifOp>
                        | <infixOp> | <mixfixOp>
                        | <prefixOp> | <postfixOp>
                        | <selectOp>
<constant>          ::= (id <name>)
<numeral>           ::= (const <name>)
<plainOp>           ::= (id <name>)
<ifOp>              ::= (if)
<infixOp>           ::= (infix <name>)
<mixfixOp>          ::= (mixfix NUMBER <trueFalse> <trueFalse>
                         <name> <name>)
<postfixOp>         ::= (postfix <name>)
<prefixOp>          ::= (prefix <name>)
<selectOp>          ::= (select <name>)
<trueFalse>         ::= true | false
```

```
<opId>             ::= <id>                          ;;; lexical op[0-9]+

<operator>         ::= <literal> | <opId>
<literal>          ::= (lit <sortId> <number>)

<prop>             ::= <term> | <genPartBy> | <traitRef> | <converts>

<quantifier>       ::= {all | exists} <varId>

<extension>        ::= <scope> | <subscope>
<scope>            ::= (scope NUMBER)
<subscope>         ::= (subscope NUMBER NUMBER?)

<signature>        ::= (<sortId>*) sortId

<sortId>           ::= <id>                          ;;; lexical s[0-9]+
<sortConstructor>  ::= <sortId>

<typedef>          ::= <enumeration> | <tuple> | <union>

<enumeration>      ::= (enum <sortId> <succOp> <enumItem>+)
<enumItem>         ::= (<enumItemName> <enumItemOp>)

<tuple>            ::= (tuple <sortId> <opId> <field>+)
<field>            ::= (<id> <sortId> <opId> <opId>)
<union>            ::= (union <sortId> <sortId> <opId> <field>+)

<predicate>        ::=   <term>
<term>             ::=   <applicationTerm> | <quantifiedTerm>
                       | <referenceTerm> | <literal>
<applicationTerm>  ::=   (apply <opId> <term>+)
<quantifiedTerm>   ::=   (<quantifier> <term>)
<referenceTerm>    ::=   <varId> | <opId> | <sortId>

<trait>            ::= (trait <traitName> <traitFormals>?
                        {<shorthand> | <external>}* <opDcls>?
                        {(asserts <varDcls>? <prop>+ <extension>)}?
                        {(implies <varDcls>? <prop>+ <extension>)}?)
<traitName>        ::= <name>

<traitParam>       ::= <sortId> | <opId> | <sortConstructor>

<traitRef>         ::= (uses (traitref "<traitName>" (<traitParamActual>))

<varId>            ::= <id>                          ;;; lexical v[0-9]+

<renaming>         ::= (rename <replace>+)
<replace>          ::= (<traitParam> <traitParam>)

<renamingMap>      ::= <replace>*
<replace>          ::= (<traitParam> <traitParam>)

<action>           ::= (<actionId> <actionType> <name>
                        {<formals> <where>?}? <extension>)
```

```
<actionType>       ::= input | output | internal

<actionId>         ::= <id>                          ;;; lexical a[0-9]+

<assignment>       ::= (assign <lvalue> <value>)
<lvalue>           ::= <varId> | (apply <opId> <lvalue> <term>+)

<automaton>        ::= (automaton <automatonName> <formals>? <automatonDef>)
<automatonName>    ::= <name>
<formals>          ::= (formals <formal>+)
<automatonDef>     ::= <primitive> | <composition> | <hiding>
<primitive>        ::= ( (actions <action>+)
                         <states>
                         {(transitions <transition>+)}?
                         {(tasks <task>+)}?
                         <schedule>? )
<states>           ::= (states <soThat>? <varId>*)
<soThat>           ::= (sothat <term>)

<component>        ::= (<id> <automatonActuals>? <term>)
<automatonActuals> ::= (actuals <term>+)

<composition>      ::= (compose <component>+)

<conditional>      ::= (if ( {(<predicate> <program>)}+ ) <program>)

<simcorresp>       ::= (sim_entry <transitionId> (formals <formal>*) <program>)

<formal>           ::= <term>

<invariant>        ::= (invariant <invariantName> <automatonName> <predicate>)

<loop>             ::= (for <varId> <term> <program>)

<ndrfire>          ::= (fire {<transitionId> <actionActuals>?}?)

<ndrwhile>         ::= (while <predicate> <program>)

<ndryield>         ::= (yield <term>)

<program>          ::= (<statement>*)

<schedule>         ::= (schedule (states <state>*) <ndrprogram>)
<ndrprogram>       ::= ({<assignment> | <conditional> |
                         <ndrfire> | <ndrwhile>}*)

<simfire>          ::= (sim_fire <transitionId> <actionActuals>?
                                 (using (<variable> <term>)+)?)

<proof>            ::= (proof (states <state>*) ((<term> <value>)*)
                              (<simcorresp>*))

<simulation>       ::= (sim <direct> <automatonName> <automatonName>
                            <predicate> <proof>?)
```

```
<direct>           ::= {forward | backward}

<state>            ::= (<varId> <value>?)

<statement>        ::=  <assignment> | <conditional> | <loop> | <simfire>
                      | <ndrfire> | <ndrwhile>

<task>             ::= (task <actionSet> <forClause>? <extension>)

<transition>       ::= (<transitionId> <caseName> <actionId>
                        {<actionActuals> <where>?}?
                        <chooseFormals>? <precondition>? <effect>?)
<caseName>         ::= "<id>"?
<chooseFormals>    ::= (choose <varId>+)
<precondition>     ::= (pre <predicate>)
<transitionId>     ::= <id>                          ;;; lexical t[0-9]+

<value>            ::= <term> | <choice>
<choice>           ::= (choose {(<varId> <term>?) (<yieldprogram>)?}?)

<yieldprogram>     ::= (det <yieldstatement>*)
<yieldstatement>   ::= <conditional> | <ndrwhile> | <ndryield>
```

Appendix B

Files produced for and by Daikon

B.1 Fibonacci

Listing B-1: Fibonacci.decls

```
DECLARE
Fibonacci:::OBJECT
Fibonacci.a
int
int

Fibonacci.b
int
int

Fibonacci.c
int
int


DECLARE
Fibonacci.compute():::ENTER
Fibonacci.a
int
int

Fibonacci.b
int
int

Fibonacci.c
int
int


DECLARE
Fibonacci.compute():::EXIT
Fibonacci.a
int
```

```
int

Fibonacci.b
int
int

Fibonacci.c
int
int
```

Listing B-2: Fibonacci.dtrace, for `sim 10 -daikon Fibonacci.ioa`

```
Fibonacci:::OBJECT
Fibonacci.a
1
1
Fibonacci.b
0
1
Fibonacci.c
1
1

Fibonacci.compute():::ENTER
Fibonacci.a
1
1
Fibonacci.b
0
1
Fibonacci.c
1
1

Fibonacci.compute():::EXIT
Fibonacci.a
0
1
Fibonacci.b
1
1
Fibonacci.c
1
1

Fibonacci:::OBJECT
Fibonacci.a
0
1
Fibonacci.b
1
1
Fibonacci.c
1
1

Fibonacci.compute():::ENTER
```

```
Fibonacci.a
0
1
Fibonacci.b
1
1
Fibonacci.c
1
1

Fibonacci.compute():::EXIT
Fibonacci.a
1
1
Fibonacci.b
1
1
Fibonacci.c
2
1

Fibonacci:::OBJECT
Fibonacci.a
1
1
Fibonacci.b
1
1
Fibonacci.c
2
1

Fibonacci.compute():::ENTER
Fibonacci.a
1
1
Fibonacci.b
1
1
Fibonacci.c
2
1

Fibonacci.compute():::EXIT
Fibonacci.a
1
1
Fibonacci.b
2
1
Fibonacci.c
3
1

Fibonacci:::OBJECT
Fibonacci.a
1
1
Fibonacci.b
2
```

```
1
Fibonacci.c
3
1

Fibonacci.compute():::ENTER
Fibonacci.a
1
1
Fibonacci.b
2
1
Fibonacci.c
3
1

Fibonacci.compute():::EXIT
Fibonacci.a
2
1
Fibonacci.b
3
1
Fibonacci.c
5
1

Fibonacci:::OBJECT
Fibonacci.a
2
1
Fibonacci.b
3
1
Fibonacci.c
5
1

Fibonacci.compute():::ENTER
Fibonacci.a
2
1
Fibonacci.b
3
1
Fibonacci.c
5
1

Fibonacci.compute():::EXIT
Fibonacci.a
3
1
Fibonacci.b
5
1
Fibonacci.c
8
1
```

```
Fibonacci:::OBJECT
Fibonacci.a
3
1
Fibonacci.b
5
1
Fibonacci.c
8
1

Fibonacci.compute():::ENTER
Fibonacci.a
3
1
Fibonacci.b
5
1
Fibonacci.c
8
1

Fibonacci.compute():::EXIT
Fibonacci.a
5
1
Fibonacci.b
8
1
Fibonacci.c
13
1

Fibonacci:::OBJECT
Fibonacci.a
5
1
Fibonacci.b
8
1
Fibonacci.c
13
1

Fibonacci.compute():::ENTER
Fibonacci.a
5
1
Fibonacci.b
8
1
Fibonacci.c
13
1

Fibonacci.compute():::EXIT
Fibonacci.a
8
1
Fibonacci.b
```

```
13
1
Fibonacci.c
21
1

Fibonacci:::OBJECT
Fibonacci.a
8
1
Fibonacci.b
13
1
Fibonacci.c
21
1

Fibonacci.compute():::ENTER
Fibonacci.a
8
1
Fibonacci.b
13
1
Fibonacci.c
21
1

Fibonacci.compute():::EXIT
Fibonacci.a
13
1
Fibonacci.b
21
1
Fibonacci.c
34
1

Fibonacci:::OBJECT
Fibonacci.a
13
1
Fibonacci.b
21
1
Fibonacci.c
34
1

Fibonacci.compute():::ENTER
Fibonacci.a
13
1
Fibonacci.b
21
1
Fibonacci.c
34
1
```

```
Fibonacci.compute():::EXIT
Fibonacci.a
21
1
Fibonacci.b
34
1
Fibonacci.c
55
1

Fibonacci:::OBJECT
Fibonacci.a
21
1
Fibonacci.b
34
1
Fibonacci.c
55
1

Fibonacci.compute():::ENTER
Fibonacci.a
21
1
Fibonacci.b
34
1
Fibonacci.c
55
1

Fibonacci.compute():::EXIT
Fibonacci.a
34
1
Fibonacci.b
55
1
Fibonacci.c
89
1

Fibonacci:::OBJECT
Fibonacci.a
34
1
Fibonacci.b
55
1
Fibonacci.c
89
1
```

B.2 Hanoi

Listing B-3: Daikon output for a single Hanoi test case

```
Hanoi:::OBJECT
Hanoi.stackA[] elements != null
Hanoi.stackB[] elements != null
Hanoi.stackC[] elements != null
size(Hanoi.stackC[]) == - size(Hanoi.stackA[]) - size(Hanoi.stackB[]) + 5
===========================================================================
Hanoi.putOnA():::ENTER
size(Hanoi.stackA[]) <= 2
size(Hanoi.stackA[]) one of { 0, 1, 2 }
size(Hanoi.stackB[]) == 0  (mod 2)
size(Hanoi.stackB[]) one of { 2, 4 }
size(Hanoi.stackC[]) >= 1
size(Hanoi.stackC[]) one of { 1, 2, 3 }
size(Hanoi.stackA[]) != size(Hanoi.stackC[])
size(Hanoi.stackA[])-1 != size(Hanoi.stackC[])-1
===========================================================================
Hanoi.putOnA():::EXIT
size(Hanoi.stackA[])-1 == orig(size(Hanoi.stackA[]))
size(Hanoi.stackA[]) >= 1
size(Hanoi.stackA[]) one of { 1, 2, 3 }
size(Hanoi.stackB[]) one of { 1, 2, 3 }
size(Hanoi.stackC[]) == 1  (mod 2)
size(Hanoi.stackC[]) one of { 1, 3 }
orig(Hanoi.stackA[]) is a subsequence of Hanoi.stackA[]
Hanoi.stackB[] is a subsequence of orig(Hanoi.stackB[])
Hanoi.stackC[] is a subsequence of orig(Hanoi.stackC[])
size(Hanoi.stackA[]) != size(Hanoi.stackB[])-1
size(Hanoi.stackA[]) != size(Hanoi.stackC[])-1
size(Hanoi.stackA[]) != orig(size(Hanoi.stackC[]))-1
size(Hanoi.stackA[])-1 != size(Hanoi.stackB[])
size(Hanoi.stackB[]) != size(Hanoi.stackC[])-1
size(Hanoi.stackB[]) != orig(size(Hanoi.stackC[]))-1
size(Hanoi.stackB[])-1 != orig(size(Hanoi.stackA[]))-1
size(Hanoi.stackB[])-1 != orig(size(Hanoi.stackC[]))
size(Hanoi.stackC[]) != orig(size(Hanoi.stackB[]))
size(Hanoi.stackC[])-1 != orig(size(Hanoi.stackB[]))-1
orig(size(Hanoi.stackA[]))-1 != orig(size(Hanoi.stackC[]))-1
===========================================================================
Hanoi.putOnB():::ENTER
size(Hanoi.stackA[]) >= 1
size(Hanoi.stackC[]) == 1  (mod 2)
size(Hanoi.stackC[]) one of { 1, 3 }
size(Hanoi.stackA[])-1 != size(Hanoi.stackB[])
===========================================================================
Hanoi.putOnB():::EXIT
size(Hanoi.stackB[])-1 == orig(size(Hanoi.stackB[]))
size(Hanoi.stackA[]) == 1  (mod 2)
size(Hanoi.stackA[]) one of { 1, 3 }
Hanoi.stackA[] is a subsequence of orig(Hanoi.stackA[])
orig(Hanoi.stackB[]) is a subsequence of Hanoi.stackB[]
Hanoi.stackC[] is a subsequence of orig(Hanoi.stackC[])
size(Hanoi.stackA[]) != orig(size(Hanoi.stackC[]))-1
size(Hanoi.stackA[])-1 != orig(size(Hanoi.stackC[]))
size(Hanoi.stackB[]) != orig(size(Hanoi.stackA[]))
size(Hanoi.stackB[])-1 != size(Hanoi.stackC[])
size(Hanoi.stackB[])-1 != orig(size(Hanoi.stackA[]))-1
```

```
size(Hanoi.stackC[]) != orig(size(Hanoi.stackA[]))
size(Hanoi.stackC[])-1 != orig(size(Hanoi.stackA[]))-1
size(Hanoi.stackC[])-1 != orig(size(Hanoi.stackB[]))-1
orig(size(Hanoi.stackC[])) == - size(Hanoi.stackB[]) - orig(size(Hanoi.stackA[])) + 6
===========================================================================
Hanoi.putOnC():::ENTER
size(Hanoi.stackA[]) == 1  (mod 2)
size(Hanoi.stackA[]) one of { 1, 3, 5 }
size(Hanoi.stackB[]) != size(Hanoi.stackC[])-1
size(Hanoi.stackB[])-1 != size(Hanoi.stackC[])
===========================================================================
Hanoi.putOnC():::EXIT
size(Hanoi.stackC[])-1 == orig(size(Hanoi.stackC[]))
size(Hanoi.stackB[]) == 0  (mod 2)
size(Hanoi.stackB[]) one of { 0, 2, 4 }
size(Hanoi.stackC[]) >= 1
Hanoi.stackA[] is a subsequence of orig(Hanoi.stackA[])
Hanoi.stackB[] is a subsequence of orig(Hanoi.stackB[])
orig(Hanoi.stackC[]) is a subsequence of Hanoi.stackC[]
size(Hanoi.stackA[]) != size(Hanoi.stackC[])
size(Hanoi.stackA[]) != orig(size(Hanoi.stackB[]))-1
size(Hanoi.stackA[]) != orig(size(Hanoi.stackC[]))-1
size(Hanoi.stackA[])-1 != size(Hanoi.stackC[])-1
size(Hanoi.stackA[])-1 != orig(size(Hanoi.stackB[]))
size(Hanoi.stackB[]) != orig(size(Hanoi.stackA[]))
size(Hanoi.stackB[])-1 != orig(size(Hanoi.stackA[]))-1
size(Hanoi.stackC[]) != orig(size(Hanoi.stackB[]))
size(Hanoi.stackC[])-1 != orig(size(Hanoi.stackB[]))-1
orig(size(Hanoi.stackB[])) != orig(size(Hanoi.stackC[]))-1
orig(size(Hanoi.stackB[])) == - size(Hanoi.stackC[]) - orig(size(Hanoi.stackA[])) + 6
===========================================================================
Hanoi.start():::ENTER
size(Hanoi.stackA[]) == size(Hanoi.stackB[]) == size(Hanoi.stackC[])
Hanoi.stackA[] == []
Hanoi.stackB[] == []
Hanoi.stackC[] == []
size(Hanoi.stackA[]) == 0
===========================================================================
Hanoi.start():::EXIT
size(Hanoi.stackB[]) == size(Hanoi.stackC[]) == orig(size(Hanoi.stackA[]))
                     == orig(size(Hanoi.stackB[])) == orig(size(Hanoi.stackC[]))
Hanoi.stackA[] contains no nulls and has only one value, of length 5
Hanoi.stackB[] == []
Hanoi.stackC[] == []
size(Hanoi.stackA[]) == 5
```

Bibliography

[1] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, MIT Department of EECS, September 2001.

[2] Andrej Bogdanov, Laura Dean, and Christine Karlovich. Distributed memory algorithms coded in IOA: Challenge problems for software analysis and synthesis methods. Manuscript, January 2001.

[3] Anna E. Chefter. A simulator for the IOA language. Master's thesis, MIT Department of EECS, May 1998.

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

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.

[6] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems.* Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 2000.
`URL=http://theory.lcs.mit.edu/tds/papers/Garland/ioaManual.ps.gz`.

[7] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-

Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

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

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

[10] Shien Jin Ong. Investigating simulations between I/O automata of asynchronous memory systems. Manuscript, URL=`http://theory.lcs.mit.edu/~shienjin/paper.ps`, August 2001.

[11] J. Antonio Ramírez-Robredo. Paired simulation of I/O automata. Master's thesis, MIT Department of EECS, September 2000.

[12] Michael J. Tsai. Abstract data types for IOA code generation. Manuscript, URL=`http://theory.lcs.mit.edu/~mjt/ADTsForIOACodeGeneration.pdf`, August 2001.

[13] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. *Model Checking TLA+ Specifications*, pages 54–66. Number 1703 in Lecture Notes in Computer Science. Springer, January 1999. URL=`http://www.research.compaq.com/SRC/tla/papers.html`.