Paired Simulation of I/O Automata

by

J. Antonio Ramírez-Robredo

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

Master of Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author ....
Department of Electrical Engineering and Computer Science
August 30, 2000

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

Paired Simulation of I/O Automata

by

J. Antonio Ramírez-Robredo

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2000, in partial fulfillment of the
requirements for the degree of
Master of Electrical Engineering and Computer Science

Abstract

An important principle that permeates theoretical work in distributed systems is that of *successive refinement*. This principle encourages the algorithm designer to start with a high-level description of the system, and to successively refine it down to lower-level implementations. For this purpose, it is common to relate two automata using a *forward simulation relation*, which is a mathematical relation between the states of the automata. Given the importance of this technique, it is desirable to have software tools to work with simulation relations.

In this thesis, I describe my design of a new simulator for the IOA language with the capability of simulating pairs of automata together, which are claimed to be related by a given simulation relation. This simulator is able to verify if the proposed simulation relation holds in a particular execution of the low-level automaton, given a step correspondence between the automata. The work to accomplish this goal included designing language extensions to IOA for specifying the step correspondence (in addition to the simulation relation) between the automata, as well as a new approach for resolving single-automaton nondeterminism.

I have provided documentation on the software implementation of the simulator and necessary support modules so that future work on the IOA toolkit (be it work on the simulator or on other tools) can be done using this software environment as a foundation.

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

Acknowledgments

My advisor, Prof. Nancy Lynch, gave me expert direction in all aspects of the project, and she helped me make this thesis into a much better document than I could have done on my own.

The work of Dr. Stephen Garland on Larch and the IOA Frontend was a vital prerequisite to this thesis, and I also wish to thank him for useful conversations and insightful feedback on my ideas.

This extended simulator could not exist without Anna E. Chefter's prior work, which served as an admirable starting point and as an encouraging assurance that the goal was within reach.

Joshua A. Tauber and Michael J. Tsai gave me a vote of confidence by using the internal representation code in their own projects, and they made very useful comments. Mandana Vaziri was generous in sharing her experience with me, and I had very valuable conversations with her. Undergraduate researchers Laura G. Dean, Christine A. Karlovich, Christopher H. Luhrs, and Ezra Y. Rosen wrote interesting IOA examples which served as inspiration for work on the simulator, and as useful indicators of future simulator projects. Laura G. Dean also read drafts of my thesis and pointed out numerous mistakes. More broadly, I am thankful to the members of the Theory of Distributed Systems group, who have been most encouraging.

Finally, the quality of my experience at MIT would have been greatly diminished without the loving support from my family and the encouragement and companionship from my friends. It is thanks to them that I was able to maintain my wits and finish this work.

Para mis padres.

Contents

List of Listings

List of Figures

Chapter 1

Introduction

One of the most important research activities in the area of distributed systems is the development of mathematical tools for the formal modeling and verification of distributed algorithms. This mathematical machinery should permit a precise specification of allowable behaviors exhibited by a system, as well as applicable methods for determining the correctness of implementations. One such proposed tool is the *Input/Output Automaton* model [7], I/O automaton for short, which is a labeled transition system that allows for modular construction of concurrent systems from smaller components. This model has been influential in the distributed systems research community, and much of the work in the Theory of Distributed Systems (TDS) group has the formalism of I/O automata at its core.

1.1 Theoretical foundation

In this section I present a brief summary of the principal definitions on which the IOA Toolkit is founded. All of these definitions have been taken almost verbatim from the textbook *Distributed Algorithms* by Nancy A. Lynch [7]. The description below is terse, and the reader is referred to this textbook for a more detailed discussion of the definitions and their motivations.

1.1.1 I/O Automata

A *signature* S is a triple consisting of three disjoint sets of actions:

- $in(S)$, the *input actions*,
- $out(S)$, the *output actions*,
- $int(S)$, the *internal actions.*

In terms of these components we also define:

- $local(S) := out(S) \cup in(S)$, the *locally controlled actions*,
- $acts(S) := in(S) \cup inter(S) \cup out$, all the actions.

An *input/output automaton* A (I/O automaton for short) consists of five components:

- $sig(A)$, a signature,
- $states(A)$, a (not necessarily finite) set of *states*,
- $start(A) \subseteq states(A)$, a nonempty set, known as the *start* or *initial* states of A.
- a set $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$ of *transitions* of A, with the property that for every state s and every input action π there exists a transition $(s, \pi, s') \in trans(A)$.
- $tasks(A)$, a partition $local(sig(A))$ into at most countably many classes.

1.1.2 Executions and traces

An *execution fragment* of an I/O automaton A is either a finite sequence $s_0, \pi_1, s_1, \ldots, s_r$, or an infinite sequence $s_0, \pi_1, s_1, \ldots, \pi_r, s_r, \ldots$, of alternating states and actions of A such that $(s_k, \pi_{k+1}, s_{k+1})$ is a transition of A for each $k \geq 0$. If s_0 is a start state of A, then the execution fragment is called an *execution.* A state of A is said to be *reachable* if it is the final state of a finite execution of A. The *trace* of an execution fragment α, denoted by $trace(\alpha)$, is the subsequence of α consisting of all external actions, and it represents the externally-observed behavior of A during the execution of α.

1.1.3 Simulation Relations

A *forward simulation relation* (or just simulation relation) from automaton A to automaton B is a binary relation $f \subseteq states(A) \times states(B)$ such that:

1. If $s \in start(A)$, then $f(s) \cap start(B) \neq 0$.

2. If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and $(s, \pi, s') \in trans(A)$, then there is an execution fragment α of B starting with u and ending with some $u' \in f(s')$, such that $trace(\alpha) = trace(\pi)$,

where $f(s)$ stands for $\{u : (s, u) \in f\}$. Simulation relations are an important tool in the study of distributed systems, and their relevance stems from the following:

Theorem. If there is a simulation relation from A to B, then $traces(A) \subseteq traces(B)$.

In other words, the existence of the simulation relation shows that A implements B. Not every trace inclusion can be proved using forward simulation relations. For this reason, there exist further variants of this definition; see Lynch and Vaandrager [8] for a number of them. In this document, I will only consider forward simulation relations.

1.2 The IOA Toolkit and its motivation

Along with the abstract mathematical tools, it is highly desirable to have as a counterpart to the theory a set of software tools to aid with the processes of analysis and implementation of algorithms. An ongoing project at the TDS group is the creation of the *IOA Toolkit*, a suite of software tools that address these concerns. My own work forms part of this toolkit.

At the core of the IOA Toolkit is the programming language IOA, which closely shadows I/O automata in notation and semantics. The language is described in [3], [4]. IOA inherits several properties from the I/O automaton model that make it an unusual programming language; for example, rather than having an explicit flow of control, executions are specified through actions, which may be enabled or disabled according to the current state. Multiple actions may be enabled at a given point in time, and hence this programming language is nondeterministic. Moreover, the language permits the manipulation of mathematical objects of unbounded size, a feature that contributes to its lack of orthodoxy. Needless to say, these properties raise difficult implementation issues. The motivation for these design choices is a desire to develop systems starting with the strong foundation of the I/O automaton model. Thus, rather than imposing limitations on the computational model because of implementation concerns, the approach is to enforce a high degree of closeness to an a-priori model, hence making the language simple, general, and easy to reason about. The extra generality has the tradeoff of making the implementation of some tools more involved. IOA has been specified as an application of the *Larch Shared Language* (LSL) [5], which allows the IOA toolkit to tap into the rich theorem-proving system Larch.

Throughout this document, I will assume some familiarity with the content of the IOA User and Reference Manual [3].

1.3 The IOA simulator

Anna E. Chefter designed and implemented an IOA tool, called the IOA Simulator [1]. The simulator, given an IOA automaton specification, performs a software simulation of an execution of the I/O automaton that it represents. This tool is potentially a useful aid during the design of a system using IOA, since it allows the designer to see the algorithm in action. One of the difficulties in designing a simulator for IOA is resolving the nondeterminism present in this language, in order to select an execution to simulate among all the possible ones.

My Master of Engineering thesis project consists of the following:

- Improving the design of the simulator and the mechanisms for resolution of nondeterminism.
- Extending the simulator so that it allows invariant checking. This is a simple addition once the simulator exists, since it only entails evaluating each of the invariants of an automaton after each step. See Chapters 2 and 3 for a description of my design regarding this and the previous item.
- Extending the simulator to allow *paired simulation*: given a simulation relation between two automata, and a proposed step correspondence, use an execution of the low-level automaton to induce an execution of the high-level automaton using the step correspondence, while checking the validity of the simulation relation. This is the main part of my project. See Chapter 4 for more motivation on this problem and the way in which I addressed it.
- Developing a software environment that facilitates the implementation of future extensions of the simulator, and future tools in the IOA Toolkit. An introduction to the use of this software environment is given in Chapter 6.

1.4 Notes on terminology

I would like to clarify several points regarding the terminology that is used in the rest of the document

1. Some I/O automaton-related words are used with slightly different meanings, depending on whether they refer to the abstract I/O automaton model or to syntactic elements of an automaton specification written in IOA. For instance, an IOA `transition` block typically defines a *family* of transitions of the automaton that is being modeled, one for each value of its actual parameters and for each value of the explicit choices that may occur in its effect program. A similar remark is true for automaton actions and their parameters.

2. Throughout this document, the word "simulation" can refer to one of two different notions: on one hand, it means the act of using a software program to execute one or more I/O automata described in IOA; on the other, it refers to a mathematical simulation relation between two abstract I/O automata. It is usually clear from the context which of these two meanings is intended; otherwise, I have used the term "paired simulation" to refer to the *software* simulation of *mathematical* simulation relations (and hence the title of this thesis).

3. A large part of the discussion on resolution of nondeterminism pertains to the choose keyword in IOA. According to the formal grammar, explicit choices using this keyword are classed as "value" nonterminals, but I want to avoid the ambiguity between the explicit choice as a syntactic element and the value of the choice in a particular execution, and hence I avoid the expression "choose value". An alternative expression could be "choose term", but that is inaccurate since the choose keyword is not valid in every context that a term is valid. As a compromise, I have settled for the expression "choose statement": while not a statement itself, an explicit choice can only appear as the right-hand side of an assignment, which is a statement.

4. In the context of resolution of nondeterminism, I have made use of Larch code which is syntactically correct, but whose semantics, as interpreted by the simulator, do not conform to the semantics of Larch traits; in particular, the implementation uses a pseudo-random number generator, while Larch semantics dictate a deterministic implementation. This was done for convenience only, since the mechanism for specifying Larch traits was very close to what was needed to specify the signatures of certain nondeterministic operators used by the simulator. I will refer to items defined in this way as *pseudotraits*, to emphasize the contrast with genuine Larch traits. It is plausible to eventually add an extension to IOA to allow a form of this syntax to be used by the simulator. Refer to Section 2.5 for the specifics.

Chapter 2

Resolution of nondeterminism

One of the central goals of the IOA language is that of high expressivity for mathematical modeling of distributed systems. As part of this goal, IOA incorporates a family of nondeterministic constructs. In order to simulate an automaton, a particular execution must be chosen, and perhaps the main problem to be solved in this respect is to design a satisfactory mechanism for the resolution of nondeterminism. In this chapter I outline some desirable characteristics of such a mechanism, and the way in which I propose to achieve some of them. I will use the abbreviation NDR[1] to refer to resolution of nondeterminism.

2.1 The problem

There are several of sources of nondeterminism in the IOA language; for instance:

- an automaton can have multiple enabled actions in a given state,
- a given enabled action can have multiple transition definitions associated with it,
- a given transition definition can take arbitrary actual parameter values, as long as they satisfy its where clause, and,

[1] **n**on**d**eterminism **r**esolution

- a transition definition can contain one or more choose statements, each of which may evaluate to an arbitrary value that satisfies the constraint in the where clause.

From the point of view of an IOA automaton specification, the sources of nondeterminism can be all regarded together as a black box that can yield both transitions to be scheduled and values to be assigned to choose statements in transitions. Thus the problem of resolution of nondeterminism can be regarded as that of providing an algorithmic means of obtaining these values and transitions as the need for them arises during the simulation of an automaton. To make the simulator a useful tool, it is desirable to make this mechanism:

- *Broad.* It should provide several ways to resolve nondeterminism, each suited to different situations and applications. For instance, it should allow choices and transitions to be resolved as deterministic functions of the automaton's state, or using a pseudo-random number generator, or by querying the user, or any combination of these.

- *Extensible.* It should be sufficiently open-ended that future developers and advanced users can tailor it to specific needs without too much effort. For instance, if a new datatype implementation is added to the simulator, it should be possible to add useful NDR mechanisms to go with it.

- *Usable.* It should be reasonably easy to use, and it should not place cumbersome demands upon the user. It is my opinion that this is the most important of the three points: resolution of nondeterminism is an absolute necessity for nontrivial uses of the simulator, and it would be unfortunate that a lack of attention to usability considerations should discourage its use.

2.2 A previous approach to NDR

Anna E. Chefter designed an NDR mechanism for IOA automaton specifications [1]. The essentials of her approach are as follows: for each automaton to be simulated there must exist a *determinator* specification. To handle `choose` nondeterminism, the determinator provides for each `choose` statement in the automaton a finite set from which its values are to be drawn. The simulator then uses these values to resolve the choice, selecting them uniformly at random. To handle transition nondeterminism, the determinator contains what amounts to a sequential program. This program is presented as a series of if/then statements, which specify the transitions to schedule. Thus it allows the specification of behavior like: "if any automaton [in some composition] has more than fifty messages in its buffer, then give it priority to take a step" [1]. Additionally, determinators provide a way to generate pseudo-random numbers and query the user for values to be used as parameters to transitions. Besides serving as an NDR mechanism, Chefter's determinator also has the ability to annotate scheduled transitions with simulated timing and "weight" information.

2.3 Motivation for a new approach

My own approach to NDR has several similarities with Anna E. Chefter's method. For example, the mechanism for resolution of automaton transitions is also presented as a sequential program that can, among other things, use if/then rules to yield transitions as a function of the automaton's state. However, the determinator framework, as described in [1], has some drawbacks, which I sought to address:

1. In determinators, choices can only be resolved using a pseudo-random number generator. Thus this mechanism is restrictive, since the value of a `choose` statement may be consistently correlated with, say, a state variable of the automaton, and this might be a useful case to simulate. A second problem is that the set of admissible values for a given `choose` statement may vary during an execution, which is a situation not addressed by this mechanism.

2. A determinator specification has the advantage of being neatly separated from the automaton itself. In particular, this property yields the benefit that no changes to the IOA grammar itself are necessary in order to implement them.

Unfortunately, it also has the side effect of requiring each choose statement in the automaton to have a unique name for its dummy variable, so that it can be unambiguously referenced in the determinator. This is undesirable, since it requires the designer to enforce this global uniqueness of names, which can become a source of errors.

3. When there are multiple transition definitions with the same signature, it is necessary to distinguish among them in some way when referring to them. Determinators accomplish this by using the sequential position of the transition definitions in the automaton. For instance, the determinator can distinguish between "output myAction:[1]" and "output myAction:[2]", if there are two transition definitions for the same output action with name "myAction". While this does the job, there is the the possibility of modifying the automaton by altering the order of the transitions, inadvertently changing the meaning of the determinator. This reliance of the determinator semantics on the syntactic specifics of the automaton can make it easy for the user to make mistakes, since localized changes with no semantic effects on the automaton (e.g., reordering the transitions) can modify the semantics of the determinator.

4. The schemes for user interaction and random number generation are fixed as part of the determinator syntax. For example, there is no method to produce pseudo-random reals between 0 and 1, and, more importantly, there is no way to add such a method without modifying the determinator grammar itself.

The ways in which I address these points are as follows:

1. My proposed NDR scheme allows arbitrary rules for determining each choose statement. These rules are described as sequential programs which can make decisions based on the evaluation of arbitrary IOA terms.

2. My proposal requires the programmer to augment the automaton specification itself with NDR-related information. Namely, it requires a schedule block for resolving automaton transitions, and a det block for resolving the values of each

choose statement. Thus, it needs additions to the IOA grammar itself. However, since the NDR information is syntactically local to the explicit choices, no global unique-naming constraint is necessary.[2]

3. In order to address the problem of disambiguation among transition definitions with the same signature, my proposal adds syntax to IOA for explicitly naming the transition definitions themselves. For example, now an automaton can have two transition definitions output myAction case A and output myAction case B. The token after the new case keyword can be an arbitrary identifier or a numeral, and it is used in the schedule and det blocks to refer to specific transition definitions. In this way, a permutation of the transition definitions does not affect the assignment of the case names, leaving the semantics of the schedule block intact.

4. The schedule and det blocks can evaluate arbitrary IOA terms to decide which transitions to schedule, or which values to yield for a choice. In addition, they can evaluate operators whose implementations perform pseudo-random number generation, or user prompting, to yield a result. This has the advantage that the NDR mechanism can be extended in essentially the same way that new datatypes are added to the simulator, as described in Chapter 6.

2.4 Overview of the proposed NDR mechanism

Generally speaking, my approach to NDR is to assign a program, called an *NDR program*, to each source of nondeterminism in an automaton. Each such program is capable of providing values that resolve a choice, or transitions to be scheduled, depending on the context. Thus there is an NDR program corresponding to every choose statement in an automaton, and an NDR program for scheduling the actions of the automaton. In this section I will illustrate the operation of the NDR mechanism

[2]Contrast this with the situation in paired simulation, in Chapter 4, in which uniqueness of dummy variable names *is* required in the specification-level automaton in a simulation relation.

using simple examples. Refer to Chapter 3 for a more detailed and general description of the interpretation of NDR programs by the simulator.

Listing 2-1: `Chooser.ioa`

```
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
```

Listing 2-2: `Chooser.ioa`, with NDR

```
automaton Chooser
  signature
    output action1, action2(n:Int)
  states
    chosen: Int
  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
```

Consider the IOA code in Listing 2-1. It is an artificial example that exhibits nondeterminism both from choices and from transitions. It contains a transition which is always enabled and whose effect nondeterministically chooses a value to assign to the single state variable. A second transition definition has a parameter, and it is enabled only when the state variable equals its parameters. This specification can be augmented with NDR programs to resolve its nondeterminism, for example, as shown in Listing 2-2. This example contains the basic features of my approach. Notice these crucial points:

- The NDR program in the schedule block uses the fire statement to schedule transitions of the automaton, hence "firing" them. This statement allows the specification of the type of action (input, output, external) and its parameters, which may in turn depend on the values of state variables of the automaton.

Similarly, the NDR program associated with the choose statement uses the yield statement to specify the values of the choice.

- The NDR program associated with the choose statement has three successive yield statements. The semantics are as follows: when the simulator encounters the choose statement, it will start executing the NDR program until it encounters a yield statement. At this point, it will use the value provided by the statement as the value of the choose statement, and it will remember the current statement of the NDR program. The next time it encounters the same choose statement, the simulator will *not* start its NDR program from the beginning; rather, it will resume executing it where it left off. Thus, in the example in Listing 2-2, the choice will be resolved successively to 1, 2, and 3. Similarly, in the schedule block, the simulator will remember where it left off after a transition was fired and resume from there the next time it schedules a transition.[3]

- Moreover, in the case of choose statements, there is an implicit infinite loop surrounding the statements of the NDR program. Because of this, the choose statement in the example resolves to the values 1, 2, 3, 1, 2, 3, etc. This convention is not used in the case of the schedule block in the automaton, but the same effect can be obtained by explicitly writing an infinite loop, as shown.

  There is a rationale for these design choices. I expect it to be common for a given choose statement to be resolved in the same way each time it is encountered: say, by invoking the same pseudo-random number generator, by prompting the user in the same manner, or by computing the same deterministic function. This is reflected in the NDR program as an infinite loop around a statement, which would be impractical for users to specify manually if this is indeed a common case. This is not a desirable convention in the schedule block: many automata do not have infinite executions, and for them, one must be able to express schedules that eventually stop producing transitions and halt. An implicit infinite loop

[3]The semantics of yield and fire statements were inspired by the iterator construct in the programming language CLU. [6]

would disallow this.

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

Listing 2-3 shows the output of the simulator on this automaton. The simulator takes as command line parameters the number of transitions to simulate, the name of the automaton to simulate, and the name of a file containing the intermediate language form of the IOA specification. For every step taken by the automaton (including the initialization step), the simulator reports the transition that was executed, and the state variables that changed.

Listing 2-3: Simulator output on Chooser automaton.

```
% java ioa.simulator.shell.SimShell 5 Chooser Chooser.il
[[[[ begin initialization [[[[
     EVENT: initialized simulator
%%%% Modified state variables:
     chosen --> 0
]]]] end initialization ]]]]
[[[[ begin step 1 [[[[
     EVENT: transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 1
]]]] end step 1 ]]]]
[[[[ begin step 2 [[[[
     EVENT: transition: output action2(1) in automaton Chooser
%%%% No modified state variables
]]]] end step 2 ]]]]
[[[[ begin step 3 [[[[
     EVENT: transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 2
]]]] end step 3 ]]]]
[[[[ begin step 4 [[[[
     EVENT: transition: output action2(2) in automaton Chooser
%%%% No modified state variables
]]]] end step 4 ]]]]
[[[[ begin step 5 [[[[
     EVENT: transition: output action1 in automaton Chooser
%%%% Modified state variables:
     chosen --> 3
]]]] end step 5 ]]]]
     No errors

%
```

2.5 Other NDR features

There are a few important aspects of the NDR mechanism that are not illustrated by this example:

1. A schedule NDR program can fire input actions. This is provided for convenience, since otherwise it would be necessary to compose the automaton with an environment automaton in order to provide a full schedule. Once a satisfactory mechanism for simulation of compositions is in place, this feature might not be as important.

2. It is sometimes desirable to resolve choices and schedule transitions using pseudo-randomness or user input as information. This issue can be addressed by providing extra operators that evaluate as random number generators and user prompters. One way to do this is to use a pseudotrait[4] such as the one in Listing 2-4.

Listing 2-4: `NonDet.lsl`

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

Each of these operators is either currently implemented by the simulator, or is easy to implement with the current software support. Using them, an alternative way of resolving the nondeterminism of Chooser is as follows:

[4]I would like to emphasize that LSL traits are meant to model *deterministic* mathematical operators, and that therefore this is not an orthodox use of Larch. For example, a zero-ary operator in an LSL trait represents a constant, and does not admit implementations that evaluate differently at different times. Because of this, the NonDet "trait" is not meant to be used in a general IOA context (e.g., inside an effect block); rather, it was introduced to be used only in NDR programs. This turns out to be a convenient and flexible way to incorporate these capabilities. For example, users of the simulator can extend the NDR capabilities in the same way that they can add implementations of specialized operators, as described in Chapter 6. I use the term "pseudotrait" to refer to this and other objects that are syntactically like LSL traits, but whose implementations by the simulator do not conform to LSL semantics. A possible future expansion would be to expand the syntax of IOA to allow declarations similar to the NonDet pseudotrait without abusing the semantics of LSL.

Listing 2-5: `Chooser.ioa`

```
uses NonDet

automaton Chooser
  signature
    output action1, action2(n:Int)
  states
    chosen: Int
  transitions
    output action1
      eff chosen := choose x where 1 <= x /\ x <= 30
                      det do
                            yield randomInt(1,30)
                          od
    output action2(n)
      pre n = chosen
  schedule do
    while true do
      fire output action1;
      if randomBool then fire output action2(chosen) fi
    od
  od
```

In a similar way, user prompting can be used instead of randomness.

3. There are circumstances in which it would be tedious to write a complete schedule by hand, and in which the simulator by itself can find an appropriate transition to schedule. The mechanism supports the statement fire with no arguments. When the simulator encounters a statement of this kind in an NDR context, it will:

   (a) examine in turn each locally-controlled transition definition of the automaton among those whose actual parameters are constants. For each of them, evaluate the precondition to see if it is enabled, and,

   (b) among those that are enabled, choose one uniformly at random and fire it in the usual way.

4. In IOA, multiple transition definitions can share the same action type, name and actual parameter sorts. In this scenario, the form of the fire statement shown in Listing 2-2 would be ambiguous. This problem is solved by using the case keyword in the transition definition to specify a name; it can be used, for example, as in Listing 2-6. The case name of the transition is local to the primitive automaton in which it is defined, and it can be a number or an alphanumeric identifier. Using the case identifier, the NDR program in the schedule block can distinguish between the two transitions.

Listing 2-6: `Undecided.ioa`

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

2.6 Future work

The above-described syntax for resolution of nondeterminism, while flexible, might be regarded as requiring too much work for its use. For example, it demands that the user provide an NDR program associated with each choose statement in an automaton, which could result in repetitive code fragments scattered over the automaton's code, or in a schedule block that is too complex. Here I present some possible future additions to the current NDR syntax that would help remedy this to some extent. None of the extensions discussed here have been implemented, and the following chapters are independent of this section.

2.6.1 Per-sort choose NDR programs

One natural extension of the choice resolution syntax is the ability to specify a default NDR program associated to a given sort. This kind of feature would require a small effort to implement. One approach is to use syntactic sugar,[5] making code like the following:

[5]Strictly speaking, this transformation would not be "syntactic" sugar, since its implementation requires static semantic information on the types of choose variables.

```
automaton A
  signature
    internal doThing
  states
    x,y : Int
  transitions
    internal doThing
      eff  x := choose ;
           y := choose det do yield 3; yield 4 od
  choosing
    for w:Int do yield randomInt(-100,100) od
```

stand for:

```
automaton A
  signature
    internal doThing
  states
    x,y : Int
  transitions
    internal doThing
      eff  x := choose det do yield randomInt(-100,100) od ;
           y := choose det do yield 3; yield 4 od
```

Thus, the choosing block would specify default NDR programs for sorts used in the automaton; these programs could be overridden by NDR programs explicitly provided in choose statements. Another possibility is to allow a global choosing block, besides per-automaton ones. A mechanism of this style would perhaps go a long way towards avoiding the repetitive NDR specification of many common cases. An adequate development of this idea would include a specification of the semantics in the case that some of the choose statements have where clauses.

2.6.2 Per-predicate choose NDR programs

In a given choose construct, of the general form

choose x **where** P(x),

it could be advantageous to associate a default NDR program not with the sort of the choose statement, but with the predicate P. For example, many common integer-valued choose statements have where predicates that restrict the range of the chosen value to some fixed finite set S of numbers, such as an interval. It is redundant, given a collection of choose statements restricted in this way, to specify NDR programs for each of them, if all that is wanted is to choose, say, a random element of S each time the choice is encountered. With the current mechanism, reasonable programs can result in repetitive NDR-augmented IOA code such as the following:[6]

```
automaton A
  ⋮
  transitions
    ⋮
    eff  y1 := choose x where 0 ≤ x ∧ x ≤ 20
                        yield randomInt(0,20) ;
         y2 := choose x where 10 ≤ x ∧ x ≤ 30
                        yield randomInt(10,30) ;
         y3 := choose x where -6 ≤ x ∧ x ≤ 28
                        yield randomInt(-6,28) ;
         y4 := choose x where -20 ≤ x ∧ x ≤ 20
                        yield randomInt(-20,20)
```

Each NDR program is essentially a repetition of the where clause, and it amounts to giving the simulator a license to select the value of the choose randomly from the corresponding interval. A way to avoid this repetition is to develop a language extension similar to the following:

```
automaton A
  ⋮
  transitions
    ⋮
    eff  y1 := choose x where 0 ≤ x ∧ x ≤ 20 ;
         y2 := choose x where 10 ≤ x ∧ x ≤ 30 ;
         y3 := choose x where -6 ≤ x ∧ x ≤ 28 ;
```

[6]This example also illustrates some syntactic sugar currently supported by the syntax: the construct "choose... yield t" is equivalent to "choose... det do yield t od". See Chapter 5 for the detailed grammar.

```
      y4 := choose x where -20 ≤ x ∧ x ≤ 20 ;
  ⋮
  choosing
    for q:Int where p:Int ≤ q ∧ q ≤ r:Int
      do yield randomInt(p,r) od
```

The choosing block would associate NDR programs to families of predicates, specified in the form of patterns, which can appear as where clauses. An implementation of this feature is likely to require typed pattern matching, and perhaps to place restrictions on the types of predicates P that are acceptable in this mechanism. I believe this is a promising extension, since it would result in the power to develop an "NDR library" of useful where predicates in choices (e.g., real and integer intervals, finite sets, primes, etc.) with a variety of methods to resolve their nondeterminism, and the user would only have to select the NDR method from this library. It would be plausible to make the feature described in Section 2.6.1 a special case of this mechanism.

2.6.3 Per-task schedule NDR programs

A natural extension to the schedule block is to allow a separate NDR program for each task of the automaton. This raises the questions of how to allocate execution steps among the provided programs, and what to do in the case when some of the tasks are parameterized.

2.6.4 Articulating simulability conditions

It would be useful to define a suitable set of syntactically-specified "simulability conditions" for IOA automaton specifications. These conditions should be narrow enough such that an automaton that satisfies them can be executed with few user-specified NDR decorations, or none at all; they should also be broad enough that writing IOA specifications satisfying the conditions is not difficult, and possible for many interesting cases. This is clearly an open-ended problem. In this section, I will discuss a few possible simulability conditions, in decreasing order of restrictiveness.

1. Disallow logical quantifiers, choose and for statements, and disallow automata that have actions with formal parameters. This permits an easy algorithm for simulation: iterate through the transition definitions, and for each of them evaluate the corresponding precondition; this can be done easily due to the absence of quantifiers. Choose a transition (perhaps randomly) among those whose precondition evaluates to true, and execute it.

2. As in 1, but allow formal parameters in actions, as long as they only appear as constant values in transition definitions. For example, a transition definition headed by "output act(1,false)" is allowed, but not one headed by "output act(n:Int, b:Bool)". This is essentially the same as 1, and it makes the automaton easy to simulate for the same reasons.

3. As in 2, but allow arbitrary formal parameters in actions, as long as transition definitions are restricted to the 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
   - $\mathit{restPred}$ is an IOA predicate (without quantifiers).

   This has the result that at most one value of the variables satisfies the precondition in a given state. In this way, the simulator can, for each transition, evaluate each term_i and verify whether $\mathit{restPred}$ holds after the var_i have been substituted by the results of evaluating the term_i.

4. As in 3, but relax the restriction on the form of transitions as follows:

$$
\begin{array}{l}
\mathit{actionType}\ \mathit{actionName}(\mathit{var}_1 : \mathit{sort}_1, \mathit{var}_2 : \mathit{sort}_2, \ldots, \mathit{var}_n : \mathit{sort}_n) \\
\quad \textsf{pre}\ \ \mathit{term}'_1 = \mathit{term}_1 \wedge \\
\qquad\quad \mathit{term}'_2 = \mathit{term}_2 \wedge \\
\qquad\qquad\quad \vdots \\
\qquad\quad \mathit{term}'_n = \mathit{term}_n \wedge \\
\qquad\quad \mathit{restPred} \\
\quad \textsf{eff}\ \ldots
\end{array}
$$

where

- each term'_i is of the form $op_i(var_i, t_{i,1}, \ldots, t_{i,r_i})$, 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).

This sort of automaton would be simulable, provided that the scheme for operator implementations presented in Chapter 6 is extended to allow certain operators to implement a *search* operation. Given values $a_{i,j}$ and c of the appropriate sorts, this operation would yield a value b_i, if one exists, such that $op_i(b_i, a_{i,1}, \ldots, a_{i,r_i}) = c$. This operation might not be implemented (or it might be impossible to implement efficiently) for all operators, and only those for which it is implemented would be allowed in this context.

The simulator could then, for each transition, evaluate each term_i, invoke the appropriate *search* operation on the implementation of operator op_i, and if the search operation is valid for all i, evaluate the predicate *restPred* after substituting each of the results of the search operations into their corresponding var_i. Subsequently, the simulator can then select a transition definition with parameters among those for which this test is successful.

It is clear that further relaxations of these simulability conditions are conceivable, especially in the presence of the *search* operation. For example, this operation would allow the evaluation of a class of existential quantifiers. I have the impression that restriction 3, along with versions of the NDR extensions proposed in the preceding paragraphs, would go a long way towards making the simulator easy to use.

It would be useful to investigate simulability conditions that are preserved by Anna E. Chefter's composer algorithm ([1]; see also remarks in Section 3.4.1). A satisfactory simulability condition that is preserved by the composer algorithm would greatly expand the scope of the simulator, since it would reduce the amount of necessary user exposure to the output of the composer, which is possibly more difficult to understand.

Chapter 3

Single-automaton simulation

In this chapter I describe how the simulator is designed, both regarding the IOA language support that it requires, and the algorithm that it follows to simulate an automaton. Additionally, I present examples of how the simulator is used. I do not treat details such as the management of operator and sort implementations. For more information regarding this and other software-related issues of the simulator, refer to Chapter 6.

3.1 Limitations of the simulator

The current implementation of the simulator has the following limitations:

1. No existential or universal quantifiers are permitted anywhere in the IOA automaton to be simulated. Often, the effect of an existential quantifier can be achieved using a suitably constrained `choose` statement, as described in [1], thereby reducing the problem of evaluating such quantifiers to the problem of nondeterminism resolution for `choose` statements. Evaluating universal quantifiers would require an essentially different mechanism.

2. There are restrictions on the actual parameters in transition definitions: each of them must be either a pure variable, or a term that contains no variables, so that it evaluates to a constant. Again, as explained in [1], this is not a

real restriction, since expression parameters can be replaced by variables that are suitably constrained by the where clause of the transition. It would not be difficult to modify the current implementation to remove this constraint, but some corresponding changes to the NDR mechanisms would be necessary. I did not investigate this possibility.

3. No for loops are permitted anywhere in the automaton to be simulated. The IOA for construct is very powerful and nondeterministic. There is no straightforward way to reduce its execution to the problem of choose determination, and it probably requires specialized language extensions and support from the type implementations. Additionally, the presence of choose statements inside the body of a for loop would raise serious questions regarding determination.

4. The simulator only supports primitive automaton specifications. Chefter's thesis [1] describes a transformation algorithm (the *composer*) which takes an IOA automaton composition specification as an input, and results in an IOA specification of a primitive automaton that is equivalent to it. This kind of algorithm, if implemented, would be usable with this simulator, assuming that the user is willing to provide the necessary NDR programs for the output of the composer. See Section 3.4.1 for further discussion on this approach to simulating composite automata.

3.2 The simulator algorithm

A good way to understand how the simulator interprets the NDR programs is through a description of the algorithm that it follows. Page 36 contains a pseudo-code description of this algorithm. It is organized in three procedures. The main one is SIMULATE(A), where A is the primitive automaton specification to be simulated. This procedure in turn uses two auxiliary ones, also presented in the figure. The algorithm does not describe the details of evaluating IOA programs or terms and focuses on the NDR mechanisms. Evaluating a term requires every operator in the term to have a simulator implementation; refer to Chapter 6 for the details on matching operators and sorts with their implementations.

Notation

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

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

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

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

The procedure SIMULATE initializes a program counter for each NDR program in the automaton, including the schedule block and each choose statement. This means that the algorithm keeps a separate NDR program counter for each choose *statement*; this does *not* necessarily translate to keeping a separate program counter for each actual *choice* in the abstract automaton that is being simulated. For example, consider the following code:

Listing 3-1: ManyChoices.ioa

```
uses NonDet
automaton ManyChoices
  signature
    internal doThing(n:Int)
  states
    m: Int
  transitions
    internal doThing(n)
      eff m := choose det do yield 1; yield 2 od
  schedule do while true do
    fire internal doThing(randomInt(1,100))
  od od
```

The choose statement in the transition for doThing actually represents an infinitude of choices, one for each value of the parameter n for doThing. A single program counter is kept for all of them, and one must bear this in mind when designing NDR programs. In this case, the consequence is that this choose statement is always resolved alternatively to 1 and 2, regardless of the parameter of the transition. An alternative architecture is possible in this respect: the simulator could dynamically allocate a new NDR program counter for each new set of parameter values that it encounters for the transition, keeping all the allocated program counters in a table keyed by the parameter values. This raises concerns of memory efficiency, but is an interesting possibility.

3.3 Invariant checking

The simulator has the capability of checking whether the invariants of an automaton, stated using IOA syntax, hold throughout an execution. This is done simply by evaluating each of the invariants found in the IOA specification after each transition is executing, and issuing a warning message if any of them fail.

Listing 3-2: `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

% true invariant:
invariant A of Fibonacci:
  a + b = c

% false invariant:
invariant B of Fibonacci:
  a - b = c
```

The code in Figure 3-2 is an IOA specification of an automaton, along with two proposed invariants of its state and suitable NDR programs.[1] Figure 3-3 presents the corresponding output of the simulator; this output shows that one of the invariants did not hold on this particular execution.

Listing 3-3: Simulator output with invariant checking on `Fibonacci.ioa`.

```
% java ioa.simulator.shell.SimShell 5 Fibonacci Fibonacci.il
[[[[ begin initialization [[[[
     EVENT: initialized simulator
%%%% Modified state variables:
     c --> 1
     b --> 0
     a --> 1
]]]] end initialization ]]]]
[[[[ begin step 1 [[[[
     EVENT: transition: internal compute in automaton Fibonacci
     EVENT: invariant B failed
%%%% Modified state variables:
     c --> 1
     b --> 1
     a --> 0
]]]] end step 1 ]]]]
[[[[ begin step 2 [[[[
     EVENT: transition: internal compute in automaton Fibonacci
     EVENT: invariant B failed
```

[1] This IOA specification gives names to the invariants, using a syntax that is not part of IOA as described in [3]. See Chapter 5 for a description of this extension. It was added merely for the convenience of allowing the simulator to refer to invariants by name.

```
%%%% Modified state variables:
     c --> 2
     b --> 1
     a --> 1
]]]] end step 2 ]]]]
[[[[ begin step 3 [[[[
     EVENT: transition: internal compute in automaton Fibonacci
     EVENT: invariant B failed
%%%% Modified state variables:
     c --> 3
     b --> 2
     a --> 1
]]]] end step 3 ]]]]
[[[[ begin step 4 [[[[
     EVENT: transition: internal compute in automaton Fibonacci
     EVENT: invariant B failed
%%%% Modified state variables:
     c --> 5
     b --> 3
     a --> 2
]]]] end step 4 ]]]]
[[[[ begin step 5 [[[[
     EVENT: transition: internal compute in automaton Fibonacci
     EVENT: invariant B failed
%%%% Modified state variables:
     c --> 8
     b --> 5
     a --> 3
]]]] end step 5 ]]]]
     No errors

%
```

3.4 Future work

In this section I present areas in which the NDR mechanism described above is not entirely adequate, and outline possible directions of future improvement.

3.4.1 Simulating explicit compositions

Currently, the simulator cannot handle composite automata directly. As suggested above, it is possible to do the following:

1. Using an implementation of Chefter's composer transformation, turn an explicit IOA composition of automata into an IOA primitive automaton specification.

2. After applying the composer, manually edit its output to augment it with the NDR programs that are requisite for simulation by this implementation.

While this approach would certainly work, it raises some usability considerations. For one, it is reasonable that users of the simulator will want to specify the nondeterminism resolution directly in the vocabulary of an explicit composition. Running it through the composer results in an equivalent automaton that, however, is likely to be more complex than the original input. Moreover, the composer will introduce extra variables and transitions that might obscure the function of the original automaton, making the task of specifying the NDR programs more difficult. I would like to stress the importance of usability: simulating an I/O automaton is not conceptually a hard problem, and it would be a mistake to make a simulator that is unnecessarily difficult to use. In this light, I think that an interesting direction of future research is to extend the simulator so that it can deal directly with (perhaps restricted) explicit compositions.

3.4.2 Graphical user interface

Graphical user environments have become the norm for complex software that requires user interaction. The simulator could benefit greatly from a well-designed graphical interface. For example, it could be possible to:

1. Graphically represent the state of the automata being simulated.

2. Use dialog boxes to query the user when necessary for NDR purposes.

3. Allow the user to change the NDR parameters of the automata in between simulations, without manually going back to the source and front-end.

4. Allow the user to select which invariants should be checked and which should be ignored.

The simulator's software design already has some mechanisms that should be useful for a graphical user interface implementor; for example, it allows "listeners" to be registered. A listener is a Java object which is notified whenever an event occurs in the simulator. For example, the failure of an invariant and the execution of a transition are some of the events that can be handled in this way. A listener could, in particular, use the event notification to update the graphical display in response to particular types of events. This event/listener architecture could be further refined to allow a given listener to receive only a particular subset of the events. Refer to Chapter 6 for more information.

Chapter 4

Paired simulation

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

$$traces(B) \subseteq traces(A).$$

To prove a statement of this form, it is almost inevitable to use an argument by induction over the length of an execution of B. This inductive reasoning on automaton executions has been abstracted, yielding the method of simulation relations. Using this method, one seeks to construct a simulation relation f from B to A, as described in Section 1.1.3.

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

In this chapter, I describe how the simulator described in Chapter 3 was extended to allow simulation of a pair of automata related by a mathematical simulation relation. The central problem here is this: the simulation relation itself, being merely a predicate that relates the states of two automata, is not sufficient to specify how each step in the implementation automaton corresponds to a sequence of steps in the specification automaton. In general, there might be multiple step correspondences that realize a given valid simulation relation between automata, and even if there is only one, it can be difficult to find it. From this point of view, the problem of deriving a specification-level execution from an implementation-level execution is analogous to that of deriving a deterministic execution of a single automaton from a specification that allows nondeterminism. Not surprisingly, the problem of programmatically specifying a step correspondence admits a similar solution.

Related work Jonsson, Pnueli, and Rump [9] define a new technique for proving trace inclusions between abstract transition systems. The method consists in defining a *transducer*, which takes as input an execution of the implementation-level system and outputs a corresponding execution of the specification-level system. They prove a soundness theorem for this method, which states, in essence, that the existence of a correct transducer between the systems implies trace inclusion. While the transducers used in [9] are mathematical constructs, this idea suggests doing software simulation of a pair of automata while verifying a step correspondence.

4.1 A language for encoding step correspondences

A step correspondence needs to specify, for a given low-level transition, a high-level execution fragment such that the simulation relation holds between the respective final states of the transition and the execution fragment. Thus, a step correspondence can be seen as a "attempted proof" of the simulation relation, missing only the reasoning that shows that the simulation relation is preserved. To specify the proposed proof of a simulation relation, I extended the current syntax of the forward simulation IOA construct to include a new section called proof,[1] for specifying the step correspondence. This section contains one entry for each possible transition definition in the low-level automaton, and each entry encodes an algorithm for producing a high-level execution fragment, using a program similar to the NDR programs used in automaton schedule blocks. In addition to these entries, the proof section also contains an initialization block, which specifies how to set the variables of the high-level automaton given the initial state of the low-level automaton, and an optional states section that declares auxiliary variables used by the step correspondence.

Figure 4-1 shows the general high-level structure of a simulation proof encoded using this language. Note that this syntax extends the syntax for forward simulation relations in IOA. Some of the sections in the proof block have a more flexible syntax than is depicted here, and some can be omitted; refer to Chapter 5 for the detailed grammar. The states block introduces auxiliary variables used in the proof, and their initial values. The initially block specifies how to initialize the state variables of the specification automaton as a function of the implementation automaton's initial state, so as to satisfy the simulation relation.

Each $proofEntry_i$ is either the keyword ignore or a *proof program*, surrounded by do and od delimiters, according to the grammar rules for `SimProofProgram` as detailed in Chapter 5. Such a program is essentially an NDR program, of the form allowed in an automaton's schedule block, except that the fire statements must now provide additional information to resolve the choose statements of the specification automaton. If a proof program is present, the simulator will execute it from beginning to end to produce a high-level execution fragment for that case, using the fire statements to schedule transitions in the specification automaton. A proof entry equal to ignore is equivalent to a proof program with no statements, and it is used to represent an

[1]It was Dr. Stephen Garland who suggested calling the step correspondence a "proof", and making it a new part of the simulation relation definition; my original idea was to append the correspondence to the low-level automaton, which would not have been as clean a solution. It is plausible to further extend this syntax to include a complete proof, in a form suitable for automated proof verification.

forward simulation
  from *autImpl* **to** *autSpec* :
  simPredicate
  proof
    states
      $auxVar_1$: $sort_1$,
      $auxVar_2$: $sort_2$,
      ⋮
      $auxVar_m$: $sort_m$
    initially
      v_1 := $term_1$;
      v_2 := $term_2$;
      ⋮
      v_n := $term_n$
    for $actType_1$ $actName_1(actFormals_1)$
        case $caseId_1$
      $proofEntry_1$
    for $actType_2$ $actName_2(actFormals_2)$
        case $caseId_2$
      $proofEntry_2$
    ⋮
    for $actType_p$ $actName_p(actFormals_p)$
        case $caseId_p$
      $proofEntry_p$

Figure 4-1: Syntax of step correspondence.

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

Figure 4-2: fire statements in proof blocks.

empty high-level execution fragment.

The fire statements allowed in proof programs have the structure depicted in Figure 4-2. This general fire statement has the meaning: "schedule the transition of type *actionType*, name *actionName* with actual parameters *actionActuals*, using the values of the terms $term_1, \ldots, term_n$ to resolve the choose statements in the transition's effect having dummy variables $v_1, \ldots, v_n$". If present, the *caseId* label is used to disambiguate between transition definitions with the same signature.

This design imposes a constraint not present in the single-automaton case: it must be required that, for a given transition definition in the specification automaton, the choice statements in it have dummy variable names which are distinct. While in general it is undesirable to place unique-naming constraints for local dummy variables, I justify this design decision by arguing that, in the case or paired simulation, these are not just dummy variables, but serve also as natural names for the choices in a high-level transition. An alternative design would be to add syntax for explicitly naming the choice statements.

4.2 An illustrative example of paired simulation

Listing 4-1 contains a simple IOA specification containing a simulation relation with everything necessary for it to be executed. The automaton GreeterSpec is a specification for automata that produce the output action hello any number, perhaps infinite, of times. The automaton FiniteGreeter is a specialization of this automaton that only produces a finite (but arbitrary) number of hello outputs. FiniteGreeter has exactly one choice point, which occurs in its initialization of the maxGreets variable. To be able to simulate it, I provided an NDR program for it, consisting of the program that yields a random integer in the range $1 \ldots 100$ as the value of the choice.[2] A point to notice here is that the choose statement in GreeterSpec's transition definition has a dummy variable even though it does not have a where clause constraining it; this is

[2]Note that the semantics of FiniteChooser allow it to output *any* finite number of hello actions; the addition of the yield does not change these semantics: it merely modifies the behavior of the simulator, in this case by having it choose a random number in that particular range.

necessary if the simulation proof is to refer to it by name. This is another necessary IOA grammar change for paired simulation, and is described in Chapter 5.

Listing 4-1: `Greeters.ioa`: A simple simulation relation with step correspondence.

```
uses NonDet

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

automaton FiniteGreeter
  signature
    output hello
  states
    maxGreets: Int := choose yield randomInt(1,5),
    count: Int := 0
  transitions
    output hello
      pre count < maxGreets
      eff count := count + 1

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

Listing 4-2 is the output of the paired simulator on this IOA specification. As in the case of non-paired simulation, it outputs the transitions taken and state variables modified for every step of the implementation automaton. In addition, it outputs the transitions of the specification automaton induced by each implementation step. For each transition taken in either automaton, the simulator outputs the variables that were changed by the transition's effect. The absence of simulator error messages in the output indicates that the simulation relation was verified to hold, in this particular run, with this proposed step correspondence.

Listing 4-2: Paired simulator output on `Greeters.ioa`.

```
% java ioa.simulator.shell.PairedShell 5 FiniteGreeter GreeterSpec Greeters.il
[[[[ begin initialization [[[[
     EVENT: initialized simulator
%%%% Modified state variables for impl automaton:
     count --> 0
     maxGreets --> 2
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] end initialization ]]]]
[[[[ begin implementation step 1 [[[[
```

```
     Executed impl transition: transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 1
     Executed spec transition: transition: output hello in automaton GreeterSpec using true for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> true
]]]] end implementation step 1 ]]]]
[[[[ begin implementation step 2 [[[[
     Executed impl transition: transition: output hello in automaton FiniteGreeter
%%%% Modified state variables for impl automaton:
     count --> 2
     Executed spec transition: transition: output hello in automaton GreeterSpec using false for sg
%%%% Modified state variables for spec automaton:
     stillGoing --> false
]]]] end implementation step 2 ]]]]
[[[[ begin implementation step 3 [[[[
     EVENT: execution ended
]]]] end implementation step 3 ]]]]
>>>> No errors

%
```

4.3 The paired simulator algorithm

As I did in Chapter 3 for the single-automaton case, here I present pseudocode for the paired simulator. The pseudocode is organized into several procedures, of which SimulatePair is the main one.

Notation

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

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

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

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

⋆ EXECSPECEFFECT(R, s, t):
[R: IOA simulation relation
s: a **fire** statement of the form given in Figure 4-2
t: the transition of $R.spec$ corresponding to s]
assign $actionActuals$ to the formal parameters of t
if $eval(t.pre) = true$ **and** $eval(t.where) = true$ **then**
  execute the statements in $t.eff$ following IOA semantics;
  when a **choose** statement c needs to be evaluated, **call** EVALSPECCHOICE(R, s, t, c)
else
  halt with an error ∎

⋆ EVALSPECCHOICE(R, s, t, c)
[R: IOA simulation relation
s: a **fire** statement of the form given in Figure 4-2
t: the transition of $R.spec$ corresponding to s
c: a **choose** statement in $t.eff$]
let $r := eval(term_i)$, where v_i is the name of $c.var$
assign r to $c.var$
if $eval(c.where) = false$ **then**
  halt with an error ∎

⋆ CHECKSIMREL(R)
[R: IOA simulation relation]
if $eval(R.pred) = false$ **then**
  halt with an error ∎

The procedure SIMULATEPAIR invokes the algorithm for single-automaton execution described in Chapter 3, except that it calls procedure EXECCORRESPONDING for every low-level transition t that is scheduled. The procedure ExecCorresponding follows the proof program associated with t in the proof block of the simulation relation, executing each of the high-level transitions determined by fire statements. In addition, ExecCorresponding verifies that the induced high-level transitions have the same trace as t, and calls CheckSimRel to determine if the simulation relation holds at the end of the step. The procedure ExecSpecEffect, called by ExecCorresponding for each high-level transition, executes the effect program of the transition as in the single-automaton case, except that procedure EvalSpecChoice is called for every explicit choice. The latter procedure evaluates a choose statement using the value provided in the using part of the fire statement that determined the high-level transition, provided that it satisfies the where predicate.

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

4.4 Example 1: mutual simulation of simple communication channels

This example is drawn from [7], in which a version of it is used to illustrate basic ideas about simulation proofs.[3] Listing 4-3 is an IOA specification of two channel automata, together with two simulation relations between them, one in each direction. Both automata have the same external signature, with an input action send(n:Nat)

[3] In the textbook, the automaton corresponding to Channel2 is modeled as a composition of two copies of automata similar to Channel1.

and an output action receive(n:Nat). The parameters to these transitions represent messages. Automaton Channel1 uses a sequence with a first-in, first-out discipline to hold messages "in transit" in the channel. Automaton Channel2 uses two queues, and has an additional internal action transfer(n:Nat) to move messages from the first queue to the second. This code includes schedule blocks for both automata, and proof blocks for both simulation relations. The schedule NDR programs use pseudo-random numbers to generate the various actions. Both simulations simply state that the concatenation of the queues of Channel2 equals the queue in Channel1, and the step correspondences are, as expected, straightforward. Listing 4-4 shows the output of the paired simulator on the simulation from Channel2 to Channel1, and Listing 4-5 shows the output on the opposite simulation.

Listing 4-3: `Channels.ioa`

```
uses NonDet

automaton Channel1
  signature
    input send(n:Nat)
    output receive(n:Nat)
  states
    queue: Seq[Nat] := {}
  transitions
    input send(n:Nat)
      eff queue := n -| queue
    output receive(n:Nat)
      pre len(queue) ~= 0 /\
          last(queue) = n
      eff queue := init(queue)
  schedule
    do while true do
      if randomBool then
        fire input send(randomNat(1,100))
      fi ;
      if randomBool /\ len(queue) ~= 0 then
        fire output receive(last(queue))
      fi
    od od

automaton Channel2
  signature
    input send(n:Nat)
    output receive(n:Nat)
    internal transfer(n:Nat)
  states
    queue1: Seq[Nat] := {},
    queue2: Seq[Nat] := {}
  transitions
    input send(n:Nat)
      eff queue1 := n -| queue1
    internal transfer(n:Nat)
      pre len(queue1) ~= 0 /\
          last(queue1) = n
      eff queue2 := n -| queue2;
          queue1 := init(queue1)
    output receive(n:Nat)
      pre len(queue2) ~= 0 /\
          last(queue2) = n
      eff queue2 := init(queue2)
  schedule
```

```
    do while true do
      if randomBool then
        fire input send(randomNat(1,100))
      fi ;
      if randomBool /\ len(queue1) ~= 0 then
        fire internal transfer(last(queue1))
      fi ;
      if randomBool /\ len(queue2) ~= 0 then
        fire output receive(last(queue2))
      fi
    od od

forward simulation from Channel2 to Channel1 :
  Channel1.queue = Channel2.queue1 || Channel2.queue2
  proof
    initially
      queue := Channel2.queue1 || Channel2.queue2
    for input send(n:Nat) do
      fire input send(n)
    od
    for output receive(n:Nat) do
      fire output receive(n)
    od
    for internal transfer(n:Nat)
      ignore

forward simulation from Channel1 to Channel2 :
  Channel1.queue = Channel2.queue1 || Channel2.queue2
  proof
    initially
      queue1 := Channel1.queue;
      queue2 := {}
    for input send(n:Nat) do
      fire input send(n)
    od
    for output receive(n:Nat) do
      fire internal transfer(n);
      fire output receive(n)
    od
```

Listing 4-4: Paired simulator output on `Channels.ioa` (Channel2 implementing Channel1).

```
% java ioa.simulator.shell.PairedShell 10 Channel2 Channel1 Channels-proof.il
[[[[ begin initialization [[[[
     EVENT: initialized simulator
%%%% Modified state variables for impl automaton:
     queue2 --> []
     queue1 --> []
%%%% Modified state variables for spec automaton:
     queue --> []
]]]] end initialization ]]]]
[[[[ begin implementation step 1 [[[[
     Executed impl transition: transition: input send(3) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue1 --> [3]
     Executed spec transition: transition: input send(3) in automaton Channel1
%%%% Modified state variables for spec automaton:
     queue --> [3]
]]]] end implementation step 1 ]]]]
[[[[ begin implementation step 2 [[[[
     Executed impl transition: transition: input send(90) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue1 --> [90, 3]
     Executed spec transition: transition: input send(90) in automaton Channel1
%%%% Modified state variables for spec automaton:
     queue --> [90, 3]
]]]] end implementation step 2 ]]]]
[[[[ begin implementation step 3 [[[[
     Executed impl transition: transition: internal transfer(3) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue2 --> [3]
     queue1 --> [90]
]]]] end implementation step 3 ]]]]
[[[[ begin implementation step 4 [[[[
```

```
     Executed impl transition: transition: input send(56) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue1 --> [56, 90]
     Executed spec transition: transition: input send(56) in automaton Channel1
%%%% Modified state variables for spec automaton:
     queue --> [56, 90, 3]
]]]] end implementation step 4 ]]]]
[[[[ begin implementation step 5 [[[[
     Executed impl transition: transition: internal transfer(90) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue2 --> [90, 3]
     queue1 --> [56]
]]]] end implementation step 5 ]]]]
[[[[ begin implementation step 6 [[[[
     Executed impl transition: transition: output receive(3) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue2 --> [90]
     Executed spec transition: transition: output receive(3) in automaton Channel1
%%%% Modified state variables for spec automaton:
     queue --> [56, 90]
]]]] end implementation step 6 ]]]]
[[[[ begin implementation step 7 [[[[
     Executed impl transition: transition: internal transfer(56) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue2 --> [56, 90]
     queue1 --> []
]]]] end implementation step 7 ]]]]
[[[[ begin implementation step 8 [[[[
     Executed impl transition: transition: output receive(90) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue2 --> [56]
     Executed spec transition: transition: output receive(90) in automaton Channel1
%%%% Modified state variables for spec automaton:
     queue --> [56]
]]]] end implementation step 8 ]]]]
[[[[ begin implementation step 9 [[[[
     Executed impl transition: transition: input send(39) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue1 --> [39]
     Executed spec transition: transition: input send(39) in automaton Channel1
%%%% Modified state variables for spec automaton:
     queue --> [39, 56]
]]]] end implementation step 9 ]]]]
[[[[ begin implementation step 10 [[[[
     Executed impl transition: transition: input send(66) in automaton Channel2
%%%% Modified state variables for impl automaton:
     queue1 --> [66, 39]
     Executed spec transition: transition: input send(66) in automaton Channel1
%%%% Modified state variables for spec automaton:
     queue --> [66, 39, 56]
]]]] end implementation step 10 ]]]]
>>>> No errors

%
```

Listing 4-5: Paired simulator output on `Channels.ioa` (Channel1 implementing Channel2).

```
% java ioa.simulator.shell.PairedShell 10 Channel1 Channel2 Channels-proof.il
[[[[ begin initialization [[[[
     EVENT: initialized simulator
%%%% Modified state variables for impl automaton:
     queue --> []
%%%% Modified state variables for spec automaton:
     queue2 --> []
     queue1 --> []
]]]] end initialization ]]]]
[[[[ begin implementation step 1 [[[[
     Executed impl transition: transition: input send(11) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [11]
     Executed spec transition: transition: input send(11) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue1 --> [11]
]]]] end implementation step 1 ]]]]
```

```
[[[[ begin implementation step 2 [[[[
     Executed impl transition: transition: output receive(11) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> []
     Executed spec transition: transition: internal transfer(11) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue2 --> [11]
     queue1 --> []
     Executed spec transition: transition: output receive(11) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue2 --> []
     queue1 --> []
]]]] end implementation step 2 ]]]]
[[[[ begin implementation step 3 [[[[
     Executed impl transition: transition: input send(92) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [92]
     Executed spec transition: transition: input send(92) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue1 --> [92]
]]]] end implementation step 3 ]]]]
[[[[ begin implementation step 4 [[[[
     Executed impl transition: transition: input send(87) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [87, 92]
     Executed spec transition: transition: input send(87) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue1 --> [87, 92]
]]]] end implementation step 4 ]]]]
[[[[ begin implementation step 5 [[[[
     Executed impl transition: transition: input send(44) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [44, 87, 92]
     Executed spec transition: transition: input send(44) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue1 --> [44, 87, 92]
]]]] end implementation step 5 ]]]]
[[[[ begin implementation step 6 [[[[
     Executed impl transition: transition: output receive(92) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [44, 87]
     Executed spec transition: transition: internal transfer(92) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue2 --> [92]
     queue1 --> [44, 87]
     Executed spec transition: transition: output receive(92) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue2 --> []
     queue1 --> [44, 87]
]]]] end implementation step 6 ]]]]
[[[[ begin implementation step 7 [[[[
     Executed impl transition: transition: input send(60) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [60, 44, 87]
     Executed spec transition: transition: input send(60) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue1 --> [60, 44, 87]
]]]] end implementation step 7 ]]]]
[[[[ begin implementation step 8 [[[[
     Executed impl transition: transition: input send(38) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [38, 60, 44, 87]
     Executed spec transition: transition: input send(38) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue1 --> [38, 60, 44, 87]
]]]] end implementation step 8 ]]]]
[[[[ begin implementation step 9 [[[[
     Executed impl transition: transition: output receive(87) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [38, 60, 44]
     Executed spec transition: transition: internal transfer(87) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue2 --> [87]
     queue1 --> [38, 60, 44]
     Executed spec transition: transition: output receive(87) in automaton Channel2
```

```
%%%% Modified state variables for spec automaton:
     queue2 --> []
     queue1 --> [38, 60, 44]
]]]] end implementation step 9 ]]]]
[[[[ begin implementation step 10 [[[[
     Executed impl transition: transition: input send(84) in automaton Channel1
%%%% Modified state variables for impl automaton:
     queue --> [84, 38, 60, 44]
     Executed spec transition: transition: input send(84) in automaton Channel2
%%%% Modified state variables for spec automaton:
     queue1 --> [84, 38, 60, 44]
]]]] end implementation step 10 ]]]]
>>>> No errors

%
```

4.5 Example 2: The Peterson mutual exclusion algorithm

Listing 4-6 is an IOA source file containing several elements:

- Automaton MutEx is an abstract IOA specification for a two-process mutual exclusion service. This automaton supports requests for a critical section from two users, in the form of input actions try_i, $i = 0, 1$. The automaton grants the critical section to user i by executing output action crit_i. When user i is finished with the critical section, it signals so with input action exit_i, after which the service eventually responds with output action rem_i. This response signals that the corresponding user has entered its remainder region, and may make another request for the critical section.

  The service specification guarantees mutual exclusion; that is, it guarantees that the two processes will not be granted the critical section at the same time. This is stated in the form of an invariant for the MutEx automaton. However, this guarantee holds only provided that each user has a *well-formed* interaction with the service. This means that the trace of the execution, restricted to user i, has the form (try_i, crit_i, exit_i, rem_i, try_i, $\ldots$). In other words, a user will only request the critical section if it is in the remainder section, and it will only request to exit the critical section if it is already in it. Refer to [7] for more on the terminology of mutual exclusion.

- Automaton Peterson2PMutEx is an implementation of two-process mutual exclusion that uses shared variables. This is the Peterson algorithm, and the reader is referred to [7] for a correctness proof. The IOA form of this algorithm is taken almost directly from [7], where it is presented both in a traditional sequential style and as an I/O automaton in precondition-effect style. The automaton also has a schedule block, which produces only well-formed executions. Listing 4-6 also includes an invariant for Peterson2PMutEx, asserting mutual exclusion.

- A forward simulation relation from Peterson2PMutEx to MutEx is included, along with a step correspondence in the form of a proof block. Both the simulation relation and the step correspondence are quite simple. The simulation relation simply states that the region of each user is the same for the specification and the implementation. The step correspondence ignores most low-level transitions, except those that cause a region change for a user. The latter invoke the action in the specification automaton that produces the same region change.

This file includes everything necessary to perform paired simulation between automata Peterson2PMutEx and MutEx, and the output of this is shown in Listing 4-7.

One of the intended uses of the paired simulator is the possibility of detecting when a proposed simulation relation does *not* hold. As an example of this type of use, I altered automaton Peterson2PMutEx by introducing a bug. I changed the effect of internal transition setflag_0 to set variable flag_0 to 0 instead of 1. With this modification in place, I started the paired simulator for 400 steps. With luck, the bug would be found in the execution randomly chosen by the schedule block, and the simulator would halt with an error. This did indeed happen, and the result is shown in Listing 4-8 (only the relevant parts of the output are shown).

Listing 4-6: `MutEx.ioa`: A mutual exclusion service with implementation

```
uses NonDet

% -------------------------------------------------------------------
% Automaton MutEx abstracts mutual exclusion for two
% agents sharing a resource. It assumes well-formedness
% of the inputs.

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

automaton MutEx
  signature
    input try_0, try_1      % Agent requests critical region
    output crit_0, crit_1   % Service grants critical region
    input exit_0, exit_1    % Agent exits critical region
    output rem_0, rem_1     % Agent may enter remainder region
  states
    reg_0: region := rem,
    reg_1: region := rem
  transitions
    input try_0
      eff reg_0 := try
    output crit_0
      pre reg_0 = try /\ reg_1 ~= crit
      eff reg_0 := crit
    input exit_0
      eff reg_0 := exit
    output rem_0
      pre reg_0 = exit
      eff reg_0 := rem
    input try_1
      eff reg_1 := try
    output crit_1
      pre reg_1 = try /\ reg_0 ~= crit
      eff reg_1 := crit
    input exit_1
      eff reg_1 := exit
    output rem_1
      pre reg_1 = exit
      eff reg_1 := rem

invariant A of MutEx :
  ~(reg_0 = crit /\ reg_1 = crit)   % asserts mutual exclusion

% -------------------------------------------------------------------
% Automaton Peterson2PMutEx implements the Peterson two process mutual
% exclusion algorithm. It contains a schedule block for simulation,
% which also schedules input actions.

type pcVal = enumeration of rem, setflag, setturn, checkflag, checkturn,
                            leavetry, crit, reset, leaveexit

automaton Peterson2PMutEx
  signature
    input try_0, try_1      % Agent requests critical region
    output crit_0, crit_1   % Service grants critical region
    input exit_0, exit_1    % Agent exits critical region
    output rem_0, rem_1     % Agent may enter remainder region
    internal setflag_0, setflag_1
    internal setturn_0, setturn_1
    internal checkflag_0, checkflag_1
    internal checkturn_0, checkturn_1
    internal reset_0, reset_1
  states
    turn: Int := 0,             % Takes values in {0,1} only
    flag_0: Int := 0,           % Writable by task 0 only
    flag_1: Int := 0,           % Writable by task 1 only
    pc_0: pcVal := rem,         % Writable/readable by task 0 only
    pc_1: pcVal := rem,         % Writable/readable by task 1 only
    reg_0: region := rem,
    reg_1: region := rem
  transitions
    input try_0
      eff pc_0 := setflag ; reg_0 := try
    internal setflag_0
```

```
    pre pc_0 = setflag
    eff flag_0 := 1 ;
        pc_0 := setturn
  internal setturn_0
    pre pc_0 = setturn
    eff turn := 0 ;
        pc_0 := checkflag
  internal checkflag_0
    pre pc_0 = checkflag
    eff if flag_1 = 0 then
          pc_0 := leavetry
        else
          pc_0 := checkturn
        fi
  internal checkturn_0
    pre pc_0 = checkturn
    eff if turn ~= 0 then
          pc_0 := leavetry
        else
          pc_0 := checkflag
        fi
  output crit_0
    pre pc_0 = leavetry
    eff pc_0 := crit ; reg_0 := crit
  input exit_0
    eff pc_0 := reset ; reg_0 := exit
  internal reset_0
    pre pc_0 = reset
    eff flag_0 := 0 ;
        pc_0 := leaveexit
  output rem_0
    pre pc_0 = leaveexit
    eff pc_0 := rem ; reg_0 := rem
  input try_1
    eff pc_1 := setflag ; reg_1 := try
  internal setflag_1
    pre pc_1 = setflag
    eff flag_1 := 1 ;
        pc_1 := setturn
  internal setturn_1
    pre pc_1 = setturn
    eff turn := 1 ;
        pc_1 := checkflag
  internal checkflag_1
    pre pc_1 = checkflag
    eff if flag_0 = 0 then
          pc_1 := leavetry
        else
          pc_1 := checkturn
        fi
  internal checkturn_1
    pre pc_1 = checkturn
    eff if turn ~= 1 then
          pc_1 := leavetry
        else
          pc_1 := checkflag
        fi
  output crit_1
    pre pc_1 = leavetry
    eff pc_1 := crit ; reg_1 := crit
  input exit_1
    eff pc_1 := reset ; reg_1 := exit
  internal reset_1
    pre pc_1 = reset
    eff flag_1 := 0 ;
        pc_1 := leaveexit
  output rem_1
    pre pc_1 = leaveexit
    eff pc_1 := rem ; reg_1 := rem
schedule
  states
    die: Nat
  do while true do
    die := randomNat(1,7) ;
    if die = 1 then
```

```
        if reg_0 = rem then
          fire input try_0
        elseif reg_0 = crit then
          fire input exit_0
        fi
      elseif die = 2 then
        if reg_1 = rem then
          fire input try_1
        elseif reg_1 = crit then
          fire input exit_1
        fi
      else
        fire % (fire any enabled locally-controlled transition)
      fi
    od od

invariant B of Peterson2PMutEx :
  ~(reg_0 = crit /\ reg_1 = crit)    % asserts mutual exclusion

forward simulation
  from Peterson2PMutEx
  to MutEx : MutEx.reg_0 = Peterson2PMutEx.reg_0 /\
             MutEx.reg_1 = Peterson2PMutEx.reg_1
  proof
    initially
      reg_0 := Peterson2PMutEx.reg_0 ;
      reg_1 := Peterson2PMutEx.reg_1
    for input try_0 do
      fire input try_0
    od
    for internal setflag_0
      ignore
    for internal setturn_0
      ignore
    for internal checkflag_0
      ignore
    for internal checkturn_0
      ignore
    for output crit_0 do
      fire output crit_0
    od
    for input exit_0 do
      fire input exit_0
    od
    for internal reset_0
      ignore
    for output rem_0 do
      fire output rem_0
    od
    for input try_1 do
      fire input try_1
    od
    for internal setflag_1
      ignore
    for internal setturn_1
      ignore
    for internal checkflag_1
      ignore
    for internal checkturn_1
      ignore
    for output crit_1 do
      fire output crit_1
    od
    for input exit_1 do
      fire input exit_1
    od
    for internal reset_1
      ignore
    for output rem_1 do
      fire output rem_1
    od
```

Listing 4-7: Paired simulator output on MutEx.ioa

```
% java ioa.simulator.shell.PairedShell 30 Peterson2PMutEx MutEx MutEx.il
[[[[ begin initialization [[[[
     EVENT: initialized simulator
%%%% Modified state variables for impl automaton:
     reg_1 --> rem
     reg_0 --> rem
     pc_1 --> rem
     pc_0 --> rem
     flag_1 --> 0
     flag_0 --> 0
     turn --> 0
%%%% Modified state variables for spec automaton:
     reg_1 --> rem
     reg_0 --> rem
]]]] end initialization ]]]]
[[[[ begin implementation step 1 [[[[
     Executed impl transition: input try_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_1 --> try
     pc_1 --> setflag
     Executed spec transition: input try_1 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_1 --> try
]]]] end implementation step 1 ]]]]
[[[[ begin implementation step 2 [[[[
     Executed impl transition: input try_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_0 --> try
     pc_0 --> setflag
     Executed spec transition: input try_0 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_0 --> try
]]]] end implementation step 2 ]]]]
[[[[ begin implementation step 3 [[[[
     Executed impl transition: internal setflag_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_1 --> setturn
     flag_1 --> 1
]]]] end implementation step 3 ]]]]
[[[[ begin implementation step 4 [[[[
     Executed impl transition: internal setflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> setturn
     flag_0 --> 1
]]]] end implementation step 4 ]]]]
[[[[ begin implementation step 5 [[[[
     Executed impl transition: internal setturn_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_1 --> checkflag
     turn --> 1
]]]] end implementation step 5 ]]]]
[[[[ begin implementation step 6 [[[[
     Executed impl transition: internal checkflag_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_1 --> checkturn
]]]] end implementation step 6 ]]]]
[[[[ begin implementation step 7 [[[[
     Executed impl transition: internal setturn_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkflag
     turn --> 0
]]]] end implementation step 7 ]]]]
[[[[ begin implementation step 8 [[[[
     Executed impl transition: internal checkturn_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_1 --> leavetry
]]]] end implementation step 8 ]]]]
[[[[ begin implementation step 9 [[[[
     Executed impl transition: internal checkflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkturn
]]]] end implementation step 9 ]]]]
[[[[ begin implementation step 10 [[[[
     Executed impl transition: output crit_1 in automaton Peterson2PMutEx
```

```
%%%% Modified state variables for impl automaton:
     reg_1 --> crit
     pc_1 --> crit
     Executed spec transition: output crit_1 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_1 --> crit
]]]] end implementation step 10 ]]]]
[[[[ begin implementation step 11 [[[[
     Executed impl transition: internal checkturn_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkflag
]]]] end implementation step 11 ]]]]
[[[[ begin implementation step 12 [[[[
     Executed impl transition: internal checkflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkturn
]]]] end implementation step 12 ]]]]
[[[[ begin implementation step 13 [[[[
     Executed impl transition: internal checkturn_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkflag
]]]] end implementation step 13 ]]]]
[[[[ begin implementation step 14 [[[[
     Executed impl transition: internal checkflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkturn
]]]] end implementation step 14 ]]]]
[[[[ begin implementation step 15 [[[[
     Executed impl transition: internal checkturn_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkflag
]]]] end implementation step 15 ]]]]
[[[[ begin implementation step 16 [[[[
     Executed impl transition: internal checkflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkturn
]]]] end implementation step 16 ]]]]
[[[[ begin implementation step 17 [[[[
     Executed impl transition: internal checkturn_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkflag
]]]] end implementation step 17 ]]]]
[[[[ begin implementation step 18 [[[[
     Executed impl transition: internal checkflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkturn
]]]] end implementation step 18 ]]]]
[[[[ begin implementation step 19 [[[[
     Executed impl transition: internal checkturn_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkflag
]]]] end implementation step 19 ]]]]
[[[[ begin implementation step 20 [[[[
     Executed impl transition: input exit_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_1 --> exit
     pc_1 --> reset
     Executed spec transition: input exit_1 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_1 --> exit
]]]] end implementation step 20 ]]]]
[[[[ begin implementation step 21 [[[[
     Executed impl transition: internal reset_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_1 --> leaveexit
     flag_1 --> 0
]]]] end implementation step 21 ]]]]
[[[[ begin implementation step 22 [[[[
     Executed impl transition: output rem_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_1 --> rem
     pc_1 --> rem
     Executed spec transition: output rem_1 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_1 --> rem
```

```
]]]] end implementation step 22 ]]]]
[[[[ begin implementation step 23 [[[[
     Executed impl transition: internal checkflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> leavetry
]]]] end implementation step 23 ]]]]
[[[[ begin implementation step 24 [[[[
     Executed impl transition: output crit_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_0 --> crit
     pc_0 --> crit
     Executed spec transition: output crit_0 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_0 --> crit
]]]] end implementation step 24 ]]]]
[[[[ begin implementation step 25 [[[[
     Executed impl transition: input exit_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_0 --> exit
     pc_0 --> reset
     Executed spec transition: input exit_0 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_0 --> exit
]]]] end implementation step 25 ]]]]
[[[[ begin implementation step 26 [[[[
     Executed impl transition: internal reset_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> leaveexit
     flag_0 --> 0
]]]] end implementation step 26 ]]]]
[[[[ begin implementation step 27 [[[[
     Executed impl transition: output rem_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_0 --> rem
     pc_0 --> rem
     Executed spec transition: output rem_0 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_0 --> rem
]]]] end implementation step 27 ]]]]
[[[[ begin implementation step 28 [[[[
     Executed impl transition: input try_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_0 --> try
     pc_0 --> setflag
     Executed spec transition: input try_0 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_0 --> try
]]]] end implementation step 28 ]]]]
[[[[ begin implementation step 29 [[[[
     Executed impl transition: internal setflag_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> setturn
     flag_0 --> 1
]]]] end implementation step 29 ]]]]
[[[[ begin implementation step 30 [[[[
     Executed impl transition: internal setturn_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_0 --> checkflag
     turn --> 0
]]]] end implementation step 30 ]]]]
>>>> No errors
```

Listing 4-8: Paired simulator output on buggy version of `MutEx.ioa`

```
% java ioa.simulator.shell.PairedShell 400 Peterson2PMutEx MutEx BrokenMutEx.il
[[[[ begin initialization [[[[
     EVENT: initialized simulator
%%%% Modified state variables for impl automaton:
     reg_1 --> rem
     reg_0 --> rem
     pc_1 --> rem
     pc_0 --> rem
     flag_1 --> 0
     flag_0 --> 0
     turn --> 0
%%%% Modified state variables for spec automaton:
     reg_1 --> rem
     reg_0 --> rem
]]]] end initialization ]]]]
[[[[ begin implementation step 1 [[[[
     Executed impl transition: input try_0 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_0 --> try
     pc_0 --> setflag
     Executed spec transition: input try_0 in automaton MutEx
%%%% Modified state variables for spec automaton:
     reg_0 --> try
]]]] end implementation step 1 ]]]]

[... etc ...]

[[[[ begin implementation step 34 [[[[
     Executed impl transition: internal setturn_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_1 --> checkflag
     turn --> 1
]]]] end implementation step 34 ]]]]
[[[[ begin implementation step 35 [[[[
     Executed impl transition: internal checkflag_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     pc_1 --> leavetry
]]]] end implementation step 35 ]]]]
[[[[ begin implementation step 36 [[[[
     Executed impl transition: output crit_1 in automaton Peterson2PMutEx
%%%% Modified state variables for impl automaton:
     reg_1 --> crit
     pc_1 --> crit
     EVENT: invariant B failed
**** EVENT: attempted to schedule invalid transition: output crit_1 in automaton MutEx;
            reason: precondition fails
****        [This event is an error; halting]
**** EVENT: FAILED simulation relation from Peterson2PMutEx to MutEx
****        [This event is an error; halting]
]]]] end implementation step 36 ]]]]
>>>> Some errors occured during simulation
```

4.6 Future work

There are many directions in which this tool can be extended. Below are some suggestions for possible future projects.

4.6.1 Improving the step correspondence language

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

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

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

4.6.2 Interfacing with a computer-assisted theorem prover

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

4.6.3 Adding syntax for providing a complete proof

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

Chapter 5

Grammar changes for simulator-related IOA extensions

In this chapter I present grammars for the additions to IOA used by the simulator. I only present those parts of the IOA grammar that were modified; the reader can refer to [3] for the rest of the IOA grammar, and for the grammar syntax conventions used here.

5.1 Labeling of transition definitions

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

Original:

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

Modified:

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

```
                       transCase?
transCase        ::=   'case' idOrNumeral
```

5.2 Labeling of transition definitions

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

Original:

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

Modified:

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

5.3 Labeling of invariants

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

Original:

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

Modified:

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

5.4 Resolution of nondeterminism

This modification defines a way for the programmer to specify how the nondeterminism in an automaton is to be resolved by the simulator. The modification has two parts:

1. Addition of a syntax for sequential programs that specify the values to choose or the transitions to schedule ("NDR programs").

2. Extensions to the existing syntax for automaton and choose that incorporate these sequential programs.

The semantics for these changes are explained in Chapter 3.

5.4.1 Syntax for NDR programs

This grammar is very similar to the existing `program` grammar in IOA, except that it permits the new fire and yield statements, used by the NDR mechanisms to schedule automaton actions and determine values of choices, as well as the while statement, which provides a looping construct with simple deterministic semantics. Note that, for a given context in which an `NDRProgram` is accepted, only one of the two statements fire and yield is permissible. Also, assignments whose right-hand sides are chooses are not permitted, since the NDR program must be deterministic to be any use. These constraints are enforced during the static checking phase of the front-end.

```
NDRProgram       ::=  NDRStatement;*
NDRStatement     ::=  assignment
                    | NDRConditional
                    | NDRWhile
                    | NDRFire
                    | NDRYield
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'
NDRYield         ::=  'yield' term
```

5.4.2 Syntax extensions to **automaton** and **choose**

These extensions might appear more wordy than necessary. For instance, it would be possible to avoid the do...od bracketing of NDRPrograms. The reason I decided for this slightly long-winded syntax is the possibility that, in the future, additional language support mechanisms for nondeterminism resolution might be designed, and this syntax allows the head keyword (i.e., schedule or det) to still be used by these potential syntax extensions.

Extension to primitive **automaton** syntax

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

Original:

```
simpleBody      ::=  'signature' formalActionList+ states transitions tasks?
```

Modified:

```
simpleBody      ::=  'signature' formalActionList+ states transitions tasks?
                     schedule?
schedule        ::=  'schedule' states?  'do' NDRProgram 'od'
```

Extension to **choose** syntax

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

Original:

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

Modified:

```
choice          ::=  'choose' (variable ('where' predicate)?)?  choiceNDR?
choiceNDR       ::=  'det' 'do' NDRProgram 'od'
                   | NDRYield
```

5.5 Paired simulation

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

I augmented the syntax of IOA forward simulations to permit the specification of a "proof", which embodies the step correspondence. This proof specifies, for each transition that the implementation automaton might take, a way to produce a sequence of transitions for the specification automaton. These are the additions:

Original:

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

Modified:

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

Again, some front-end static checking is necessary, since this type of simulation proof only makes sense for *forward* simulations.

Chapter 6

The software environment

In this chapter I provide documentation for the Java interfaces and classes used in the implementation of the simulator and related software support. This is with the hope that future work can be done using this software environment as a basis. Through this chapter, mentions of the "IOA Toolkit distribution" refer to a software package (including source and Java executables) to be eventually made available by the Theory of Distributed Systems group, containing all of the IOA Toolkit and its documentation. The distribution is the best source of up-to-date and comprehensive documentation on the toolkit.

6.1 Review of the IOA Toolkit architecture

The IOA Toolkit is divided into a front-end and a back-end. The front-end, in general, takes IOA and LSL specifications as input, and, after checking syntax and static semantics, outputs an equivalent specification written in an intermediate language. Elements present in the intermediate language are meant to correspond rather directly with internal representations of IOA concepts that are designed to be used by IOA tools; this language is also intended to be easy to parse while still being human-readable with some effort. The current intermediate language is based on S-expressions and is very similar to the one described in [1], except for some modifications that make it more manageable in some cases. I will not present a detailed grammar of the intermediate language, since for most purposes the existing parser and internal representation can be used without having to understand it. Moreover, the intermediate syntax is likely to evolve in response to the insight that has been gained while developing IOA tools.

Technically speaking, each IOA tool is in itself a separate back-end, which takes as input the intermediate form of an IOA specification and does some tool-specific work with it. However, there is common support for the IOA tools in the form of an intermediate language parser and an internal representation of IOA elements, in the form of a Java class hierarchy. Both the parser and the internal representation hierarchy were designed to be highly flexible and reusable by IOA tools; see Section 6.5 for more details.

Since understanding the architecture of the front-end will not typically be necessary for extending the simulator or the IOA Toolkit, the following sections only describe the internal representation, the intermediate language parser, and the means of extending the simulator.

6.2 The internal representation: design basics

The goal of the internal representation is to specify and implement a set of object interfaces to be used by IOA tools. There is meant to be an interface, and a corresponding implementation, for each element that may appear as part of an IOA specification, such as automata, actions, terms, programs, and invariants.

In designing the internal representation of IOA elements, it was important to keep in mind that particular IOA tools are likely to need specialized support from the objects they use. For example, a code generator is likely to require methods to compile automaton objects, while the simulator employs methods to evaluate term objects. Moreover, it is desirable to have shared software support: for example, it would be highly impractical to need a specialized intermediate language parser for each separate IOA tool due to minor intermediate language modifications. These two goals are somewhat conflicting, since the parser will need to create objects, which will in turn have to use particular implementations of the interfaces. The solution to has two parts:

1. The elements of the internal representation are specified not with Java classes (abstract or otherwise), but with Java interfaces. These interfaces are in the Java package `ioa.il`, and their hierarchy is rooted at the interface `ioa.il.ILElement`.

2. The parser and other tool-independent support modules do not directly construct objects implementing these interfaces; rather, they use a globally-available *factory object*, which is a subclass of the `ILFactory` abstract class and has methods for constructing objects for each of the leaves in the interface inheritance hierarchy. Thus, for example, it has methods named `newPrimitiveAutomaton` and `newSimulationRelation`. This allows implementors of IOA tools to replace the global `ILFactory` with their own specializations of it, which may have domain-specific knowledge. Furthermore, a specialized `ILFactory` can recognize specialized IOA statements.[1]

There is a set of basic implementations of these interfaces, along with an implementation of `ILFactory` (the `BasicILFactory`) which constructs instances in this set. These basic implementations, as well as the `BasicILFactory`, are easy to subclass in order to add tool-specific behavior. The use of a factory object is described, for example, in [2], under the name "abstract factory pattern". See section 6.5 for concrete information on creating specialized factories.

See Figure 6-1 for the inheritance tree of the interface hierarchy used in the internal representation. For details on the interfaces, refer to the IOA Toolkit distribution, which contains the most up-to-date software and documentation.

[1] For example, this is how the simulator-specific statements yield, while, and fire were implemented.

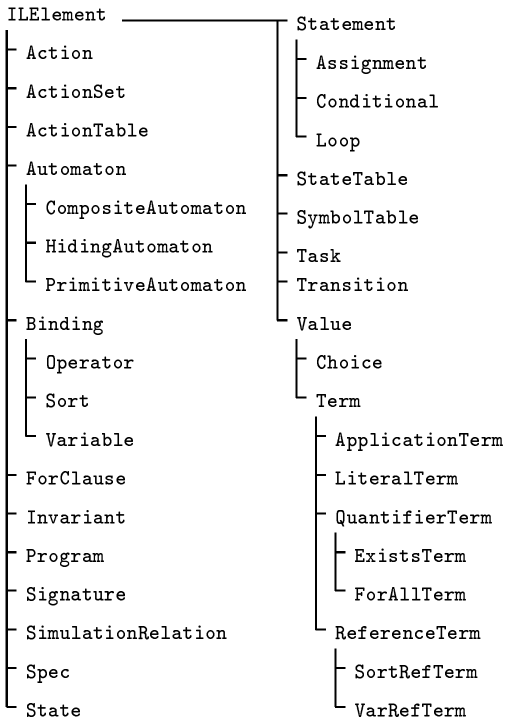

Figure 6-1: Internal representation: interface hierarchy.

6.3 The parser

The parsing of IOA specifications from the intermediate language is done by the instances of the `ILParser` class. Listing 6-1 shows an example of usage of the `ILParser`. This code sample performs the following actions, which are representative of the general usage:

1. It installs a new ILFactory, which may construct specialized implementations of the `ILElement` interfaces. This step is optional, and the `BasicILFactory` is used by default.

2. It creates an object of class `java.io.Reader`, which is the Java class used to represent streams of text. In this case, the `Reader` comes from a specific named file, but the origin is not specified.

3. It creates an `ILParser`, using the `Reader` as parameter for the constructor.

4. It invokes the method `getSpec` on the parser, which performs the actual parsing and returns an object of the interface `ioa.il.Spec`, representing the IOA specification contained in the `Reader`.

5. After this, the `Spec` object can be used according to its interface, documented in the IOA Toolkit distribution. For instance, individual automata in the specification can be obtained from the `Spec` object.

Additionally, it is necessary to use Java exception handling in the case that errors occur during parsing, in which case the `getSpec` method will throw an exception of class `ioa.il.ILParseException`. The method `getSpec` is the only external functionality available in the `ILParser`. However, the `ILParser` also allows extensive tool-specific customization, allowing it to recognize specialized IOA statements and extensions to intermediate language elements; see 6.5 for a description of these capabilities.

Listing 6-1: Example usage of the `ILParser`.

```
import ioa.il.* ;

import java.io.InputStreamReader ;
import java.io.FileInputStream ;
import java.io.File ;

// [...]

public void useParser() {
  try {
    // 1. (Optional) Install a specialized ILFactory with domain-
    // specific knowledge (by default, BasicILFactory will be used)
    ILFactory.setInstance(new MyILFactory());

    // 2. Create a Reader object (in this case, from a file)
    InputStreamReader in =
      (new InputStreamReader
        (new FileInputStream
          (new File("myFile.il"))));

    // 3. Create an ILParser with this Reader
    ILParser parser = new ILParser(in) ;

    // 4. Parse the Spec object
    Spec spec = parser.getSpec();

    // 5. Use it (for instance, get an automaton object from it)
    Automaton aut = spec.getAutomaton("myAutomaton");
  } catch(ILParseException e) {
    // 6. Handle parsing errors
    System.out.println("panic!");
  }
}
```

6.4 Adding simulator datatypes

Some applications of the simulator are likely to require support for datatypes beyond those in the current implementation. This section describes how to implement new datatypes (IOA sorts and operators), and how to have the simulator use these implementations.

Sort implementations are represented by objects implementing the interface `ioa.simulator.SortImpl`, shown in Listing 6-2. A `SortImpl` object has the necessary knowledge to create new objects of a given sort, either from scratch, or from an integer value, in the case of sorts that support numeric values. Note that, since IOA is a functional language, this is by no means the only way to construct objects: virtually every operator implementation has to create a new object to store the result it returns, as mutation is not possible.

Similarly, operator implementations are represented by objects implementing the interface `ioa.simulator.OpImpl`, in Listing 6-3. The `OpImpl` interface has a single method, `apply`, which returns the result of evaluating the operator that is being implemented on a given vector of operands. As noted above, this result will typically be a newly created object.

Both `OpImpl` and `SortImpl` handle objects implementing the interface `ioa.simulator.Entity`, shown in Listing 6-4. This is the interface for all the objects created during the simulation of an automaton. Its methods are very generic; they are enough, however, since the type-specific operations are all performed by suitable `OpImpl` objects. The name "entity" was chosen to clearly differentiate IOA-level objects from Java `Object`s.

The simulator obtains implementations for sorts and operators by querying a global *implementation registry*. This registry is an object of class `ioa.simulator.ImplRegistry`, and it contains methods that, given an operator or a sort, return a corresponding implementation. This is an abstract class, and its interface is shown in Listing 6-5. As shown, the registry method `getImpl` can return `SortImpl` and `OpImpl` objects corresponding to `Sort` and `Operator` objects, respectively.

Listing 6-2: The `SortImpl` interface.

```
package ioa.simulator ;

public interface SortImpl {
  /** This method constructs a new Entity of this sort, without
      a specified initial value */
  public Entity construct()
    throws SimException ;

  /** This method constructs a new Entity of this sort with the
      specified integral initial value (to be implemented only by sorts
      that accept literals */
  public Entity construct(int n)
    throws SimException ;
}
```

Listing 6-3: The `OpImpl` interface.

```
package ioa.simulator ;

import java.util.Vector ;

/**
 * Interface for implementations of operators.
 */
public interface OpImpl {
  /** Run the implementation code for applying the corresponding
   * operator to the given vector of operands, and return the result
   * */
  public Entity apply(Vector/*[Entity]*/ opands)
    throws SimException ;
}
```

Listing 6-4: The `Entity` interface.

```
package ioa.simulator ;

import ioa.il.* ;

/** This interface represents an object in a simulation. Entities have
 * a Sort, and are created either from scratch by SimSorts or as a
 * result of evaluating SimOperators.
 * Note: I chose "Entity" to avoid confusing these objects with java
 * "objects". */
public interface Entity {
  /**
   * Returns the sort of this entity
   */
  public Sort getSort() ;

  /**
   * Returns a string representation of this entity
   */
  public String toString() ;

  /** Returns true if and only if this entity equals the given entity,
   * in some sense depending on the particular entity.  */
  public boolean equals(Entity ent) ;
}
```

Listing 6-5: The `ImplRegistry` interface.

```
package ioa.simulator ;

import ioa.il.* ;
import java.util.Vector ;

/** This is an abstract class that represents a mapping from sorts and
 * operators to sort and operator implementation (Sorts and Operators
 * to SortImpls and OpImpls). In addition it provides static methods
 * for setting and getting the unique global instance of the
```

```
 * implementation registry.  */
public abstract class ImplRegistry {
  private static ImplRegistry instance ;

  public static ImplRegistry getInstance() { return instance ; }
  public static void setInstance(ImplRegistry newInstance) { instance = newInstance ; }

  /**
   * Returns a SortImpl for the given Sort, or null if none is known.
   */
  public abstract SortImpl getSortImpl(Sort sort)
    throws SimException ;
  /**
   * Returns an OpImpl for the given Operator, or null if none is known.
   */
  public abstract OpImpl getOpImpl (Operator operator)
    throws SimException ;
}
```

6.4.1 The `BasicImplRegistry`: an overview

For typical applications, it will not be necessary to write an implementation of `ImplRegistry` from scratch; the package `ioa.simulator.impl` contains the implementation `BasicImplRegistry`, which supports both simple sorts and parameterized sorts. This is also the default implementation registry. In this section, I describe how to add type implementations using the `BasicImplRegistry` and related classes.

See Listing 6-6 for the public interface of the `BasicImplRegistry`. Some of these methods are inherited from the superclass `ImplRegistry`, others are used for constructing a new `BasicImplRegistry`, and the remaining ones are used for installing new implementations in the registry.

When it is initialized, the `BasicImplRegistry` goes through a list of *implementation packages*, each of which is represented by a Java class (*not* a Java object). An implementation package is meant to include sorts and operators that are logically related. The package must have a static method, with the signature:

```
public static void install(BasicImplRegistry reg);
```

The `BasicImplRegistry` has a default list of implementation packages.[2] This list can be overridden by calling the `BasicImplRegistry` constructor with a Java `Enumeration`

[2] At the time of this writing, this list contains implementation packages for the sorts: Bool, Int, Nat, Array[A,B], and Seq[A]. In addition, the implementation package `ioa.simulator.impl.` `.NonDetImpl` is installed by default, and it contains implementations of some of the operators for

of strings which are fully-qualified names of implementation packages. Alternatively, the list can be overridden by setting the Java property `ioa.simulator.impl.packages` to a colon-separated list of fully-qualified package names.

For each implementation package, the `BasicImplRegistry` calls the corresponding `install` method, passing itself as the argument. This method, in turn, can call the methods `BasicImplRegistry installSortImpl`, `installOpImpl`, `installSortPreImpl`, and `installOpPreImpl`. The former two are used to install simple sorts and associated operators, while the later two are used to handle parameterized sorts. The following subsections explain the use of each of these methods.

Listing 6-6: The public interface of the `BasicImplRegistry` class.

```
package ioa.simulator.impl ;

import ioa.il.* ;
import ioa.simulator.* ;


public class BasicImplRegistry extends ImplRegistry {
  // Methods inherited from ImplRegistry
  public SortImpl getSortImpl(Sort _sort)
    throws SimException ;
  public OpImpl getOpImpl(Operator _op)
    throws SimException ;

  // Constructors
  public BasicImplRegistry()
    throws SimException ;
  public  BasicImplRegistry(Enumeration/*[String]*/ packages)
    throws SimException ;


  // Methods for installing implementations, from implementation package
  // install methods.
  public void installSortPreImpl(String name, boolean isLiteral, BasicSortPreImpl preImpl) ;
  public void installOpPreImpl(String name, BasicOpPreImpl preImpl) ;
  public void installSortImpl(String key, boolean isLiteral, BasicSortImpl impl) ;
  public void installOpImpl(String key, BasicOpImpl impl) ;

  // Auxiliary methods for installing implementations
  public static String makeOpKey(String name,String range,String[] domain) ;
  public static String makeOpKey(String name,String range) ;
  public static String makeSortKey(String name) ;
} ;
```

Simple sorts: `installSortImpl` and `installOpImpl`

A *simple* sort is one that does not take other sorts as parameters. For example, the built-in sorts Int and Nat are simple sorts. An implementation package installs a

randomness and user interaction in the Larch pseudotrait NonDet, as described in Chapter 3. (This is an example of an implementation package that is not tied to a single IOA sort or sort constructor.)

simple sort by including a line of the form

```
reg.installSortImpl(reg.makeSortKey(sortName),
                    isLiteral,
                    sortImpl);
```

in its `install` method, where:

- `reg` is the `BasicImplRegistry` object passed to the `install` method,
- `sortName` is the name of the simple sort, given as a string,
- `isLiteral` is a boolean, which is true if this sort can be assigned values specified as numerals in the IOA source[3], and
- `sortImpl` is an object of class `BasicSortImpl`, which is the sort implementation itself. This class is an implementation of the `SortImpl` interface, with some extra methods used internally by the `BasicImplRegistry`. It is often convenient to provide this argument by anonymously subclassing `BasicSortImpl`.

For example, this is the code used to install the implementation of the built-in sort Int:

```
reg.installSortImpl
  (reg.makeSortKey("Int"),
   true,
   new BasicSortImpl()
   {
     public Entity construct() { return new IntEntity() ; }
     public Entity construct(int n) { return new IntEntity(n) ; }
   }) ;
```

An operator whose signature involves only simple sorts is installed with a line of the form

```
reg.installOpImpl(reg.makeOpKey(opName,range,domain),
                  opImpl);
```

where:

- `reg` is the `BasicImplRegistry` object,

[3]This is likely to become obsolete, since the mechanism for handling literal values will probably change to become more general.

`"name"`	a plain operator (example: `"succ"`)
`"__name"`	a prefix operator (example: `"__!"`)
`"name__"`	a postfix operator (example: `"~__"`)
`"__name__"`	an infix operator (example: `"__+__"`) (mixfix operators are specified similarly, using `__` as a placeholder for arguments)
`"@<sel>field"`	a selection operator for a field (IOA syntax: a.field)

Operator symbols are encoded as described in [3]; for example, the symbol "$\in$" is encoded as `\in`, which is written as the Java string `"\\in"`.

Figure 6-2: Conventions for names of operators in implementation packages.

- `opName` is the name of the operator, given as a string,
- `domain` is the name of the range sort, given as a string,
- `range` is a tuple of the names of the domain sorts, given as an array of strings,
- `opImpl` is an object of class `BasicOpImpl`, which is an implementation of the `OpImpl` interface. Again, this is conveniently provided using Java anonymous subclassing.

The name of the operator must follow the conventions in Figure 6-2. As an example, the following code installs the greater-than-or-equal operator for the built-in sort Int:

```
reg.installOpImpl
  (reg.makeOpKey("__>=__",
                 "Bool",
                 new String[] { "Int", "Int" }),
   new BasicOpImpl()
   {
     public Entity apply(Vector/*[Entity]*/ opands)
       {
         IntEntity ent1 = (IntEntity) opands.elementAt(0) ;
         IntEntity ent2 = (IntEntity) opands.elementAt(1) ;
         return BoolEntity.make(ent1.n >= ent2.n) ;
       }
   }) ;
```

Parameterized operator and sort implementations

The `OpImpl` and `SortImpl` mechanisms described above are not sufficient to specify implementations in full generality, for these reasons:

- IOA supports the notion of a *sort constructor*, which is essentially a family of sorts parameterized by other sorts. For example, Seq is a sort constructor, since Seq[A] is a sort representing a sequence of elements of any sort A.

- There are families of operators, all with the same name and number of parameters, which have essentially the same implementation, but different signatures. For example, the equality operator $=$ takes two entities and determines if they are equal. It should be possible to specify a single implementation for all equality operators, regardless of the sort of entity that it does comparisons between. This is because the implementation can simply call the `equals` method in the `Entity` interface, without knowing the details of the particular `Entity` implementation.

This problem is addressed by providing a *preimplementation*. A preimplementation is an object which provides implementations for sorts (or operators) in some family. Given an IOA sort, in the form of a `Sort` object `s`, a sort preimplementation `p` determines if `s` belongs to the family represented by `p`, and, if so, `p` returns an appropriate `SortImpl` object for `s`. Operator preimplementations have a similar behavior. Sort and operator preimplementations are represented by the `BasicSortPreImpl` and `BasicOpPreImpl` abstract classes, respectively. See Listings 6-7 and 6-8 for their definitions and documentation on their member functions.

There exist more concrete implementations of each of these abstract classes:

- The abstract classes `MatchSortPreImpl` and `MatchOpPreImpl` provide a means to determine membership in the family by using an arbitrary boolean predicate.

- The abstract classes `TemplateSortPreImpl` and `TemplateOpPreImpl` are subclasses of `BasicSortPreImpl` and `BasicOpPreImpl` which can perform pattern matching. An example sort pattern is Sort1[Sort2[p, q],p], where Sort1 and Sort2 are names of IOA sort constructors, and p, q are variables which can match arbitrary IOA sorts.

I will not provide the source for any of these two specializations, and instead will show their usage through examples. Refer to the IOA Toolkit distribution for the source. The `Template` preimplementation classes are the ones that are likely to be used most extensively, due to their generality, and I will explain them to more depth in the next paragraphs.

Listing 6-7: The `BasicSortPreImpl` abstract class.

```
package ioa.simulator.impl ;

import ioa.simulator.* ;

/**
 * This class represents a family of sort implementations.  The method
 * getSortImpl returns a SortImpl for a given Sort, provided that this
 * sort matches (belongs to) this family; otherwise, it returns null.
 *
 * Concrete subclasses of this class must provide an implementation
 * for the construct method. There are two versions of this method.
 * The first version supports a parameter data, of type Object, which
 * provides implementation-dependent data generated during the
 * matching process.  The second version does not include this
 * parameter, and is to be used by subclasses which ignore this data.
 * */
public abstract class  BasicSortPreImpl {
  /**
   * Given a sort which matches this preimplementation, construct
   * an Entity of this sort.
   * Default implementation defers to construct(SimSort)
   *
   * @param fullsort The sort being used, which matched this preimplementation
   * @param data Implementation-specific data produced by the matching process
   */
  public  Entity construct(SimSort fullsort,Object data)
    throws SimException
  {
    return construct(fullsort) ;
  }

  /**
   * Given a sort which matches this preimplementation and an integer,
   * construct an Entity of this sort initialized to this integer (for
   * sorts that support integer values).  Default implementation
   * defers to construct(SimSort,int)
   *
   * @param fullsort The sort being used, which matched this preimplementation
   * @param data Implementation-specific data produced by the matching process
   * @param n The integer to use when constructing the Entity */
  public Entity construct(SimSort fullsort, Object data, int n)
    throws SimException
  {
    return construct(fullsort) ;
  }

  /**
   * Given a sort which matches this preimplementation, construct
   * an Entity of this sort.
   * (This method is used when match data is ignored).
   */
  public  Entity construct(SimSort fullsort)
    throws SimException
  {
    throw new SimImplException("*** unimplemented construct called") ;
  }

  /**
   * Given a sort which matches this preimplementation and an integer,
   * construct an Entity of this sort initialized to this integer (for
   * sorts that support integer values).
   * (This method is used when match data is ignored).
   */
  public Entity construct(SimSort fullsort, int n)
    throws SimException
  {
    throw new SimImplException("construction from literal unsupported by sort") ;
```

```
  }

  /**
   * Returns a sort implementation for the given sort, if this PreImpl
   * can provide one. Returns null otherwise.
   */
  public abstract SortImpl getImpl(SimSort fullsort) ;

  // For chaining (used internally by the BasicImplRegistry)
  BasicSortPreImpl next = null ;
  BasicSortPreImpl last = null ;
}
```

Listing 6-8: The `BasicOpPreImpl` abstract class.

```
package ioa.simulator.impl ;

import ioa.simulator.* ;

import java.util.Vector ;

/**
 * This class represents a family of operator implementations.  The
 * method getOpImpl returns an OpImpl for a given Operator, provided
 * that this operator matches (belongs to) this family; otherwise, it
 * returns null.
 *
 * Concrete subclasses of this class must provide an
 * implementation for the apply method, and optionally the assign
 * method. There are two versions of each of these methods. The first
 * version supports a parameter data, of type Object, which provides
 * implementation-dependent data generated during the matching
 * process.  The second version does not include this parameter, and
 * is to be used by subclasses which ignore this data.
 */
public abstract class BasicOpPreImpl {
  /**
   * Given an operator which matches this preimplementation, apply it
   * to the given vector of operands.
   * Default implementation defers to apply(SimOperator,Vector)
   *
   * @param fullop The operator being applied, which matched this preimplementation
   * @param data Implementation-specific data produced by the matching process
   * @param opands The operands of the operator
   */
  public Entity apply(SimOperator fullop,Object data,Vector/*[Entity]*/ opands)
    throws SimException
  {
    return apply(fullop,opands) ;
  }

  /**
   * Given an operator which matches this preimplementation, assign to
   * to it the given value, upon evaluation with given vector of
   * operands (if this operator supports assignment).
   * Default implementation defers to assign(SimOperator,Vector,Entity)
   * (This method is used when match data is ignored).
   *
   * @param fullop The operator being assigned to, which matched this preimplementation
   * @param data Implementation-specific data produced by the matching process
   * @param opands The operands of the operator */
  public void assign(SimOperator fullop,Object data, Vector/*[Entity]*/ opands,Entity value)
    throws SimException
  {
    assign(fullop,opands,value) ;
  }

  /**
   * Given an operator which matches this preimplementation, apply it
   * to the given vector of operands.
   * (This method is used when match data is ignored).
   */
  public Entity apply(SimOperator fullop,Vector/*[Entity]*/ opands)
```

```
    throws SimException
  {
    throw new SimImplException("*** unimplemented apply called") ;
  }

  /**
   * Given an operator which matches this preimplementation, assign to
   * to it the given value, upon evaluation with given vector of
   * operands (if this operator supports assignment).
   * (This method is used when match data is ignored).
   */
  public void assign(SimOperator fullop,Vector/*[Entity]*/ opands,Entity value)
    throws SimException
  {
    throw new SimImplException("*** unimplemented assign called") ;
  }

  /**
   * Returns an operator implementation for the given operator, if
   * this PreImpl can provide one. Returns null otherwise.
   */
  public abstract OpImpl getImpl(SimOperator fullop) ;

  // For chaining (used internally by the BasicImplRegistry)
  BasicOpPreImpl next = null ;
  BasicOpPreImpl last = null ;
}
```

Templates A template is specified using an S-expression, given as a string parameter to the constructor of `TemplateOpPreImpl` or `TemplateOpPreImpl`. In the case or sorts, the S-expression can be either:

- An S-expression of the form `(name` $\mathtt{p}_1$ `...`$\mathtt{p}_n$`)`, $n \geq 0$, where `name` is a string and the $\mathtt{p}_i$ are sort templates. In this case, the template matches any sort obtained by applying the sort constructor of name `name` to any sorts $s_1, \ldots, s_n$ such that s_i matches $\mathtt{p}_i$ for all i. (The case $n = 0$ matches only simple sorts, and in this case the parentheses can be omitted.)

- An integer $k \geq 0$, denoting a variable in the pattern. This matches any sort, provided that each integer is matched to the same sort throughout the template.

For example, a sort template that matches sorts of the form Sort1[Sort2[p, q],p] is `("Sort1" ("Sort2" 0 1) 0)`. In the case of operators, the S-expression is of the form `(name (`$\mathtt{p}_1$ `...`$\mathtt{p}_n$`)` $\mathtt{p}_0$`)`, where `name` is an operator name following the conventions in Figure 6-2. It matches any operator with name `name`, whose range matches the template $\mathtt{p}_0$ and its range sorts $s_1, \ldots, s_i$ are such that s_i matches $\mathtt{p}_i$. For example, the pattern `("__=__" (0 0) ("Bool") )` matches all the equality operators.[4].

Installing operator preimplementations An implementation package installs an operator preimplementation for a family of operators by including a line of the form

```
reg.installOpPreImpl(name,preImpl);
```

where `name` is the name of the operator, following the conventions in Figure 6-2, `preImpl` is a preimplementation object and `reg` is the `BasicImplRegistry`. For example, this is the code used to install the len operator for the sort constructor Seq[A]

```
// template: ("len" (("Seq" 0)) "Int")
reg.installOpPreImpl
  ("len",
   new TemplateOpPreImpl("(\"len\" ((\"Seq\" 0)) \"Int\")")
   {
     public Entity apply(SimOperator fullop, Vector/*[Entity]*/ opands)
       {
         SeqEntity seq = (SeqEntity) opands.elementAt(0) ;
         return IntEntity.make(seq.size()) ;
       }
   }) ;
```

Installing sort preimplementations Sort implementations are installed using a call of the form

```
reg.installSortPreImpl(name,isLiteral,preImpl);
```

where `name` is the name of the sort, `isLiteral` is a boolean, and `preImpl` is the corresponding preimplementation. An example is:

```
reg.installSortPreImpl
  ("Seq",
   false,        // isLiteral
   new MatchSortPreImpl() {
    public Entity construct(SimSort fullsort)
      { return new SeqEntity(fullsort) ; }
    public boolean matches(SimSort fullsort)
      { return fullsort.getSubSorts().size() == 1 ; }
  }) ;
```

[4]One must bear in mind that the S-expression is given as a Java string parameter, and hence all the quotes and special characters inside the S-expression must be preceded by a backslash. For example, the S-expression `("__=__" ("Bool") (0 0))`, when encoded as a Java string, becomes `"(\"__=__\" (\"Bool\") (0 0))"`

This example uses a `MatchSortPreImpl`, with a predicate that tests to see whether there is exactly one subsort given to the sort constructor `Seq`.

To learn more about providing simulator implementations, I recommend examining the source of the implementation packages provided with the simulator. This source will be included in the IOA Toolkit distribution.

6.5 Specializing the internal representation

The object oriented nature of Java allows the specialization of classes through subclassing, and this facility is the main tool for specializing the internal representation. However, this still leaves the problem of how to instruct the `ILParser` to create specialized versions of `ILElement` objects, rather than objects with the default implementation. This can be done by defining a new subclass of `ILFactory`. This class contains methods for creating each of the elements of the internal representation. Once a specialized subclass of `ILFactory` is defined, it can be installed as the default global `ILFactory` using the static method `ILFactory.setInstance`. The parser uses the installed factory to create objects of the internal representation.

All of the internal representation interfaces that are leaves of the inheritance tree in Figure 6-1 have a basic implementation. For every internal representation interface named `X`, its basic implementation is the class `ioa.il.BasicX`. For most IOA tools, it will be enough to specialize the `Basic` family. For example, if a particular tool needs to have a special-purpose method called `special` in each `Automaton` object it manipulates, it can accomplish this by:

1. Defining a subclass of `BasicAutomaton`, named, say, `SpecialAutomaton`. This class will add the metod `special`.

2. Defining a subclass of `BasicILFactory`, which redefines the method `newAutomaton` so that it creates objects of class `SpecialAutomaton` instead of `BasicAutomaton`.

3. Installing this new factory as the global factory, using `ILFactory.setInstance`.

4. Calling the parser to create an internal representation of an IOA intermediate language file.

After the last step, the returned `Spec` object will contain only `Automaton` objects which are actually of class `SpecialAutomaton`, and this hypothetical tool will be able to cast them into `SpecialAutomaton` to access the method `special`.

The `ILFactory` mechanism can also be used to allow the `ILParser` to recognize custom intermediate language statements. For more information on doing this, refer to the documentation in the IOA Toolkit distribution.

6.6 Modifying the simulator user interface

I attempted to provide some mechanisms that would allow a good user interface for the automaton to be implemented independently of the simulator itself. Towards this goal, I defined interfaces for simulator *events* and *listeners*. Simulator events are Java objects representing events that occur during a simulation of an automaton in an IOA specification, and they implement the interface `SimEvent` (Lisiting 6-9). A simulator listener is a Java object implementing the interface `SimListener` (Listing 6-10), which contains methods that are called whenever a simulator event occurs. For example, a transition taken in an automaton implements the `SimEvent` interface.

Since at the time of this writing there is only one (text-based) user interface for the simulator, it is by no means clear whether this mechanism is general enough to use as a basis for, say, a graphical user interface. For example, it could be necessary to make the listeners be event-specific. However, the architecture of the simulator is such that building on this event/listener scheme will probably not be difficult.

Listing 6-9: The `SimEvent` interface.

```
package ioa.simulator ;

/**
 * An event that may be broadcast by the simulator.
 */
public interface SimEvent {
  /**
   * Returns true if this event is an error that should cause the
   * simulation to halt.
   */
  public boolean isError() ;

  /**
   * Returns a string with a human-readable description of this
   * simulator event.
   */
  public String eventDescription() ;
}
```

Listing 6-10: The `SimListener` interface.

```
package ioa.simulator ;


/**
 * An object that may receive events broadcast by the simulator
 */
public interface SimListener {
  /**
   * Handle the given event. Return false if simulator should not
   * continue, true otherwise.
   * @exception SimException if an error occurs during handling */
  public boolean handleSimEvent(SimEvent ev)
    throws SimException ;
}
```

Bibliography

[1] Anna E. Chefter. A Simulator for the IOA Language. Master of Engineering and Bachelor of Science in Computer Science and Engineering Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1998.

[2] Erich Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[3] Stephen J. Garland, Nancy A. Lynch and Mandana Vaziri. IOA: A Language for Specifying, Programming and Validating Distributed Systems. User and Reference Manual. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, December 1997.

[4] Stephen J. Garland and Nancy A. Lynch. Using I/O Automata for Developing Distributed Systems. In Gary T. Leavens and Murali Sitaraman, editors, Foundations of Component-Based Systems, pages 285-312, Cambridge University Press, 2000.

[5] Larch: Languages and Tools for Formal Specification, John V. Guttag and James J. Horning, editors, Springer-Verlag, 1993.

[6] Barbara Liskov, et al. CLU Reference Manual, Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, Cambridge, MA, October 1979.

[7] Nancy A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

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

[9] Bengt Jonsson, Amir Pnueli and Camilla Rump. Proving refinement using transduction. Distributed Computing (1999) 12: 129-149.

[10] Bill Joy, Guy Steele, James Gosling, Gilad Bracha. The Java Language Specification, Second Edition. Addison-Wesley, 2000.