Simulation of Composite I/O Automata

by

Edward Solovey

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

Aug 22, 2003

Author ____
Department of Electrical Engineering and Computer Science
Aug 22, 2003

Certified by ____
Dilsun K. Kaynar
Post-doctoral research associate, Theory of Distributed Systems Group
Thesis Supervisor

Certified by ____
Nancy Lynch
NEC Professor of Software Science and Engineering, Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Simulation of Composite I/O Automata

by

Edward Solovey

Submitted to the
Department of Electrical Engineering and Computer Science

Aug 22, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The IOA simulator is a tool that has been developed in the Theory of Distributed Systems group at MIT. This tool simulates the execution of automata described by the IOA language. It generates logs of execution traces and provides other pertinent information regarding the execution, such as the validity of specified invariants. Although the simulator supports paired simulation of two automata for the purpose of checking simulation relations, one of its limitations is its lack of support for the simulation of composite automata. A composite automaton represents a complex system and is made up of other automata, each representing a system component. This thesis concerns the addition of a capability to simulate composite automata in a manner that allows observing and debugging the individual system component automata. While there is work in progress on creating a tool that will translate a composite definition into a single automaton, the added ability to simulate composite automata directly will add modularity and simplicity, as well as ease of observing the behavior of individual components for the purpose of distributed debugging.

Thesis Supervisor: Dilsun K. Kaynar
Title: Post-doctoral research associate, Theory of Distributed Systems Group

Thesis Supervisor: Nancy Lynch
Title: NEC Professor of Software Science and Engineering, Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my supervisor, Dilsun Kirli Kaynar, for making this experience both pleasurable and productive. Her attention to and encouragement of my ideas made their fruition possible. She attentively read many copies of my paper and provided invaluable and detailed comments. Throughout this project she has never treated me as a subordinate, but always as a peer. This gave me confidence that the goals of the project were attainable.

I am grateful to Professor Lynch and Professor Garland for the helpful conversations and comments. Professor Garland and Josh Tauber were extremely open to and helpful with my modifications to the front-end of the IOA toolkit. I was lucky enough to share an office with Josh, and he readily answered a multitude of questions that came up during my work. Without the work of prior students Toh Ne Win, Laura Dean, Antonio Ramirez, Michael Tsai, and Anna Chefter my extensions of the simulator would not have been possible.

I would like to thank my parents and grandparents whose struggles and hard work in life have paved the way for every opportunity that I might have. Their unyielding support and encouragement during this project and throughout my studies in general have made them possible. I would like to thank my sister for her support, good humor, and for being a friend.

The support and encouragement of my friends is greatly appreciated. I would like to thank Trey Reyher for his attempts to proofread my thesis, and for helping me relax during stressful times. I am most grateful to Erin Treacy whose optimistic approach towards life is an inspiration. Her company, affection, and friendship are cherished.

For Roza and Copl Levshteyn

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the major research activities within the Theory of Distributed Systems Group at MIT is the development of formal methods for modeling and reasoning about distributed systems. As distributed systems may be fairly complex, it is important to be able to model them precisely and reason about them at various levels of abstraction. The input/output (I/O) automaton model constitutes the basis of the work on formal methods [KCD+02a].

1.1 I/O Automaton Simulator Overview

The I/O automaton model is a formal model for asynchronous computing. It is a labeled transition system model suitable for describing asynchronous concurrent systems [Lyn96]. An I/O automaton models a distributed system component that can interact with other system components. It is a simple type of state machine in which transitions are associated with named actions. The IOA language, a formal language based on the I/O automaton model, provides an expressive medium for precise description of a system's behavior. Descriptions of subsystems may be composed in parallel to form a composite description. Further, the language allows for description at various levels of abstraction, and provides a mechanism for relating these descriptions [Dea01].

The IOA toolkit is a suite of software tools. It facilitates the design, analysis, and development of distributed systems within the I/O automaton framework [KCD+02a]. The tools in the suite can be broken into categories of front-end and back-end. The *checker* is a front-end tool that checks the syntax and the semantics of a given IOA file, which contains the description of one or more I/O automata. If the IOA file is valid, the checker generates an intermediate language (IL) file.

This IL file is then used by the back-end tools. The back-end tools consist of a simulator, code generator, and translators to a range of representations suitable for use with some theorem provers. This paper primarily deals with the simulator.

The IOA simulator was designed by Anna Chefter [Che98], and parts of it were implemented by Antonio Ramirez [RR00]. The input to the simulator is an IL file produced by the front-end checker described above. The simulator can be used to test automata before attempting correctness proofs. A simulation that goes as expected does not prove the correctness of the automaton but does increase confidence that the automaton has been designed as intended. A simulation that does not go as expected discovers bugs in the automaton specification. The user can then modify the automaton and run the simulator again. The following two sections describe those features of the simulator that were implemented and those that were not at the start of this project.

1.1.1 Previously Implemented Features

The simulator in release 0.09 of the IOA Toolkit is capable of simulating a primitive non-parameterized automaton. In order to resolve the nondeterminism that may be present in the automaton, the simulator is capable of using a nondeterminism resolution (NDR) block. Nondeterminism and NDR blocks are discussed in detail in Section 2.1.

In order for the simulation to be useful, the user needs to be able to observe the step-by-step behavior of the automaton being simulated. The user may invoke the simulation with a variety of command line options. Further, the user has the options to display the triggered transition at every step, never display the transition, or only display the external transitions to create a trace of the automaton. Similarly, at every step the user has the option to display all of the state variables of the automaton, none of the state variables, or just those that have been modified as a result of the transition triggered at that step.

In addition to simply observing the log of the execution, the user may provide one or more invariants for the automaton. An invariant is a predicate involving the state variables of the automaton. The simulator checks the truth value of the predicate after every step of the simulation and displays an appropriate message at every step that the predicate is false. Just as above, the fact that the invariant is true at every step does not necessarily imply that its always true. It simply increases our confidence that it might be a valid invariant. However, if it is ever false, we can discard it and look for a new one.

The simulator is also capable of paired simulations [RR00], an extremely useful feature in distributed system design and debugging. Such a simulation may be beneficial when a system is designed by moving from the highest level to the lowest level in the abstraction hierarchy. In this case the user specifies two automata, a simulation relation between the automata, and a step correspondence. The step correspondence specifies a high-level execution fragment for each low-level transition, such that the simulation relation holds after the execution of any low-level transition and its corresponding high-level fragment. The simulator then checks if the relation holds at every step of the execution. This enables the user to reason about the behavioral correspondence between the automata at different levels of abstraction [KCD+02a].

1.1.2 Newly Implemented Features

One of the simulator's prior limitations was its inability to simulate composite automata. We have implemented this feature. We first concentrate on the basic case, that of a composite automaton consisting of components that are not parameterized. Even this basic case presents the necessity to resolve a new kind of nondeterminism: nondeterminism in selecting the component that will fire the next transition. A possibility for resolving such nondeterminism is an NDR block in the composite automaton. Although at the onset of this project such NDR blocks were supported for primitive automata simulations, they were not supported for composite automata simulations. Their implementation is discussed in detail in Section 2.1.4.

More often it is the case that the components of a composite automaton will have parameters. These parameters may be either type or variable parameters. Variable parameters are now supported, while type parameters remain as future work. The introduction of parameterized components in a composite automaton introduces questions regarding the instantiation of these components and ability to access them later. These issues are discussed in Section 3.2.

The possibility of automata parameters introduces the need for constant, **const**, parameters in action signatures. A constant parameter is a term in an action signature that refers to a formal parameter of the automaton, rather than a fresh variable declaration [Tau03]. These parameters are useful when the composite automaton contains multiple components based on the same original automaton. In this case some other component may want to interact with a certain subset of these components based on their formal parameters. Constant parameters allow such interaction.

For example, Figure 1.1.1 contains the partial specifications of automata *Channel* and *Process*

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(const i, const j, m:Msg)
    output receive(const i, const j, m:Msg)
...

automaton Process(n:Int)
  signature
    input receive(const n-1, const n, x:Int)
    output send(const n, const n+1, x:Int),
           overflow(const n, s:Set[Int])
...

automaton Sys
  components C[n:Int]: Channel(Int, Int, n, n+1)
                       where 1 ≤ n ∧ n < 10;
             P[n:Int]: Process(n) where 1 ≤ n ∧ n ≤ 10;
```

Figure 1.1.1: Automaton *Sys*

[Tau03]. The automaton *Sys* is a composite automaton comprised of ten Process components and nine Channel components. The Channel components facilitate the exchange of messages between the Process components. Thus when the Process component with formal parameter **n** set to 5 sends a message, only the Channel automaton with formal parameters **i** and **j** set to 5 and 6 respectively should receive this message. Constant parameters enable this interaction. They are now supported.

The following is a high level description of the features that we have implemented:

- Simulation of Composite Automata
- Schedule Blocks in Composite Automata
- Parameters in the Components of a Composite Automaton
- Constant Parameters
- Invariants for Composite Automata

1.2 Compositions

The focus of this thesis is the simulation of composite automata. Josh Tauber's work on the *composer*, a front-end tool that converts an IOA specification of a composite automaton to an IOA specification of a primitive one, is closely related to the ability to simulate composite automata directly. In this section we introduce composite automata formally, motivate their direct simulation for the purpose of distributed debugging, compare their direct simulation to the composer, and finally propose a hierarchical debugging technique that uses both the ability to simulate composite automata directly and the composer.

1.2.1 Formal Definition of Compositions

A composition creates an automaton representing a complex system from a set of individual automata representing system components. Interaction between components is achieved through output and input actions of the same name. More specifically, when a component automaton performs an action, π, all other automata that contain an action named π perform it as well. Limitations, in the form of static semantic checks, on which automata are compatible for composition do exist [Lyn96]. They are listed below:

A countable collection $\{S_i\}_i \in I$ of signatures is compatible if for all $i, j \in I, i \neq j$, all of the following hold (where `int`, `out`, `in`, and `acts` denote internal, output, input, and all actions respectively):

1. $\text{int}(S_i) \bigcap \text{acts}(S_j) = \oslash$
2. $\text{out}(S_i) \bigcap \text{out}(S_j) = \oslash$
3. No action is contained in infinitely many sets $\text{acts}(S_i)$

Internal actions are intended to be private to a component automaton and thus unobservable by other components. The first condition ensures this. Without it a certain internal action of a component might force an action in another component to be triggered. The initial internal action would thus have to be observed. The second condition requires that the sets of output actions of all components be disjoint. This ensures that at most one component automaton "controls" the performance of any given action, which is useful when comparing the trace of the stand alone automaton to its trace when it is a component of a composite automaton. The resulting primitive automaton is defined to have the following signature:

- $\text{out}(S) = \bigcup_{i \in I} \text{out}(S_i)$
- $\text{int}(S) = \bigcup_{i \in I} \text{int}(S_i)$
- $\text{in}(S) = \bigcup_{i \in I} \text{in}(S_i) \setminus \bigcup_{i \in I} \text{out}(S_i)$

The signature, states, tasks, and start states of the produced primitive automaton are vectors of the respective pieces of the component automata. Transitions are modified to allow actions with the same name to be executed simultaneously.

Upon first glance it might seem intuitive that if a component contains an output action π and another component contains an input action π, then the action π should be an internal one in the composition. The above signature renders action π as an output action in the composition. The labeling of action π as an output action allows the resulting composed automaton to later be composed with another new component containing an input action π. Had π been labeled as an internal action after the first composition, condition one of the limitations listed above would have prevented the second composition. This is not the desired behavior as it should be possible for an output action to be broadcast to more than one automaton. The behavior is also not desirable because it makes the order of composition relevant.

The initial implementation of the composite automata simulator requires that the action **where** clauses of the component automata do not have quantifiers. Allowing quantifiers would require verification by a theorem prover to ensure that an input and an output action do indeed satisfy their respective **where** clauses. Also, the component automata are required to have no hidden actions.

It can be seen from the above definitions that unless the names of the actions and state variables of the component automata identify their owner, all modularity and tractability of individual components are lost once the automata are composed. When looking at the trace of the primitive automaton, it will not be possible to discern from which component automaton the actions originated. Nor will it be possible to discern what individual automaton contributed a particular state variable to the global state when looking at the executions of the primitive automaton. Thus, although it is possible to prepend each action and state variable with an identifier for its component automaton, it would be convenient not to do this. The ability to closely monitor the behavior of individual components during a simulation of a composite automaton is useful for the purpose of distributed debugging. We analyze this topic next.

1.2.2 Distributed Debugging

Debugging distributed systems is a much more difficult task than debugging conventional, sequential programs. This is due to the fact that distributed systems are more complex, introduce the element of synchronization, and make debugging much more difficult due to the possibility of partial failures [Kun93]. For these reasons it is crucial to be able to simulate and thoroughly debug a distributed system at design time. The capability of the simulator to run composite automata will allow for trace logs that clearly identify individual components and for testing component specific invariants. This in turn will allow the user to observe and reason about individual components of the composition, thus aiding him/her in identifying the component that is causing a problem or behaving in an unpredictable manner. Further, it will clearly display to the user the communication pattern and synchronization between component automata.

The notion of global snapshots is a tool often used for distributed debugging. A global snapshot captures the state of all the processes of the system at a certain point in time [Yan92]. Although in the case of simulating a composition as a single primitive automaton it is possible to see the state of all of the variables involved, it is not clear from which of the component automata those variables originated. In a snapshot that is produced by the direct simulation of a composition, all state is available along with the information of what component automaton contributed that particular piece of the state. Once again, this can prove to be useful during the debugging of a complicated system. In theory, when creating a snapshot of a distributed system, an algorithm such as that of Chandy and Lamport [Yan92] must be used to ensure that the snapshot represents the state of all components at the same point of the execution. Since the simulator has a single point of control, we do not have to worry about ensuring this condition.

In addition to debugging once a problem has been observed, the designer of the distributed system might want to forestall problems by proving properties about the composite system. "In order to prove properties of a composed system automata, it is often helpful to reason about the component automata individually." [Lyn96] More specifically, the designer might want to see if his/her composite automaton satisfies such trace properties as safety (some particular "bad" thing never happens) and liveness (some particular "good" thing eventually happens). Because showing that each component satisfies a particular trace property implies that the composition satisfies the product trace property, it is extremely useful to be able to reason about individual components. Similarly, if the composition fails to satisfy a product trace property, the simulator will be able to help the designer identify the particular component that failed the trace property.

1.2.3 Comparison to Composer

The composer takes the IOA specification of a composite automaton as input. As output it returns the IOA specification of an equivalent primitive automaton. Thus the composer is entirely a front-end tool. As the new primitive automaton is being created, a major part of the composer concerns the creation of composite automaton variables of appropriate sorts. For example, an entirely new state table consisting of a vector combination of the component state tables must be created. The direct simulation of composite automata moves part of this burden to the back-end tool, the simulator. Instead of attempting to create an aggregate state table, the simulator simply creates an automaton object for every component which is responsible for maintaining its own state.

Another area of concern of the composer is the semantic checking of **where** clauses in transition definitions to determine how to combine output and input transitions. For example, Figure 1.2.2 contains the specification for automaton *Sys* with components *Channel* and *Process* (where P_1,P_2,P_3, and P_4 are effects programs).

When output action `send` in automaton *Process* is fired, it may trigger the first `send` transition of automaton *Channel*, the second `send` transition of automaton *Channel*, or neither. Table 1.2.1 contains a list of the possibilities.

Value(s) of x	Executed Effects Programs
6,7,8	P_4
9,10,11	P_1,P_4
12,13	P_4
14	P_2, P_4
15,16	P_2,P_3
17,18...	P_3

Table 1.2.1: Effect clauses of automaton *Sys* to be executed

The composer has to create a separate transition in the expanded automaton for every one of the cases in Table 1.2.1. In some cases this might require an undecidable semantic proof. However, in the direct simulation case, whenever output action `send` in automaton *Process* is fired, the simulator has a value for the actual parameter **x** of the transition. The simulator can now perform boolean tests on the **where** clauses (still limited to disallow quantifiers) of the input `send` transitions of automaton *Channel* to see if any should be triggered. The case of **where** clauses in **components**

```
automaton Channel
 signature
   input send(x:Int) where x>0

...
  transitions
    input send(x) where x>8 /\ x<12
      eff P1
    input send(x) where x>13 /\ x<17
      eff P2
...

automaton Process
  signature
    output send(x:Int) where x>0
...
  transitions
    output send(x) where x>15
      eff P3
    output send(x) where x>5 /\ x<15
      eff P4
...

automaton Sys
  components Channel;Process
```

Figure 1.2.2: Combination of action **where** clauses

definitions is very similar to the above and is discussed in detail in Section 3.2.

It is true that any composite simulation that may be performed using direct simulation can be performed via two steps: first the transfer from composite automaton to primitive automaton using the composer; and second, a primitive simulation of the resulting automaton. In some cases it might be easier and faster to perform the direct simulation. Also, the direct simulation provides easy traceability of components. To achieve the same traceability, the composer would have to create some kind of system of labeling the resulting states and transitions that would maintain the individuality and modularity of the components.

1.2.4 Hierarchical Debugging

It will often be helpful to reason about a system from a hierarchical, top-down perspective, varying the levels of modules to identify the source of error. For example, [Kun93] describes a system that models an airport shuttle system. The shuttle system consists of four major components - platforms (NorthEast, NorthWest, SouthEast, and SouthWest platforms). Each of these platforms is in turn made up of smaller components - tracks (TrackNorthWest, TrackMiddleWest,...).[1] A single automaton models each track component.

To debug the system following the concepts of hierarchical debugging, the designer might first want to model the system as consisting of two parts - North and South. A round of testing might reveal an error in one of these two parts. The designer will then move down a level of modules in the erroneous part and leave the properly behaving one at the highest level of modules. If part South is found to contain an error, the second round of testing will consist of three parts - SouthEast, SouthWest, and North. This process can continue until the lowest level erroneous part has been pointed out.

The combination of the composer tool and the ability to simulate composite automata directly provides an easy way to implement hierarchical debugging as described above. The composer is used to create various levels of modules. The ability to simulate composite automata is used to identify the erroneous component at a particular level of abstraction.

The composer tool can be used to create primitive automata *AutSouth* and *AutNorth* out of composite automata that consist of all of the south and all of the north component tracks respectively. The simulator then simulates the composition of *AutSouth* and *AutNorth*. If *AutSouth* is identified as the erroneous component, the composer can be used to create primitive automata *AutSouthWest* and *AutSouthEast*. The simulator then simulates the composition of *AutSouthWest*, *AutSouthEast*, and *AutNorth*. Once again, this hierarchical process can continue until either the lowest level erroneous component has been identified, or the error has been identified at the desired level of modules.

[1] Terminology note: The term "level of modules" refers to a point in the modular hierarchy. For example, the components NorthPlatform and SouthPlatform are the highest level of modules. The next lower level of the NorthPlatform module might contain NorthEastPlatform and NorthWestPlatform. The next lower level of the NorthEastPlatform might contain NE1Platform, NE2Platform, and NE3Platform.

Chapter 2

Composite Simulations

We now shift our focus to the simulator and its handling of composite automata. How is a simulation of a composite automaton different from a simulation of a primitive automaton? How should we pick the next action to fire? Will the firing of this action involve any other components? Section 2.1 discusses a new type of nondeterminism that arises in composite simulations. Section 2.2 describes how the interaction between components is handled by the simulator.

2.1 Nondeterminism

Before discussing the nondeterminism introduced by composite simulations, in Section 2.1.1 we take a look at the nondeterminism already present in primitive automata. We then propose two methods of resolving the new nondeterminism. Section 2.1.2 discusses the reuse of nondeterminism resolution procedures provided for each component. Section 2.1.3 describes how we avoid the pitfall of nondeterminism procedure looping introduced by the reuse strategy. Section 2.1.4 discusses the creation of a nondeterminism resolution procedure tailored for the composite automaton.

2.1.1 Overview of Nondeterminism

A key feature of the IOA model is nondeterminism. Nondeterminism allows systems to be described in their most general forms and to be verified considering all possible behaviors without being tied to a particular implementation of a system design [KCD+02b]. There are two types of nondeterminism in the IOA model. Explicit nondeterminism appears in the form of **choose** statements, which may appear on the right hand side of variable assignments such as:

```
schedule
 states counter:Int:=1

do
  if(Aut.total>10) then
     fire output action1
  fi;
  counter:=counter+1;
  if(Aut.ready=true) then
     fire output action2
  fi
od
```

Figure 2.1.1: Example of a terminating NDR program

```
output action1
  eff chosen := choose x: Int where 10 < x;
```

Implicit nondeterminism arises due to ambiguity in scheduling actions [KCD+02a]. Listed below are the two ways in which implicit nondeterminism may occur:

- an automaton can have multiple enabled actions in a given state; and
- a given transition definition can take arbitrary actual parameter values, as long as they satisfy its **where** clause.

The IOA simulator is a deterministic program and cannot itself resolve the nondeterminism present in the automata that it is simulating. To solve this problem we have taken advantage of the fact that from the point of view of an IOA automaton specification, resolution of nondeterminism can be regarded as a black box that can yield transitions to be scheduled and values to be assigned to statements that involve nondeterministic choice [KCD+02a]. In other words, the automaton is not aware how the nondeterminism is resolved, but as long as it is resolved, the simulation of the automaton may go forward. The simulator requires the user to provide deterministic programs that replace these black boxes. These programs are **det** and **schedule** blocks for explicit and implicit nondeterminism respectively. In the presence of these nondeterminism resolution blocks, the simulator can deterministically simulate an automaton.

Figure 2.1.1 displays a simple schedule block for the primitive automaton *Aut*. At every step of the simulation, the simulator polls this schedule block for the next action to fire. The schedule block is then executed until a **fire** invocation is returned. The next time the schedule block is polled it

```
eff chosen := choose x where 1 ≤ x ∧ x ≤ 30
      det do
        yield 1; yield 2; yield 3
      od
```

Figure 2.1.2: Sample **det** block

resumes execution at the statement immediately after the previously returned **fire** statement. For example, if at step one of the simulation the schedule block in Figure 2.1.1 returned `fire output action1`, then upon its next polling it will start execution at `counter:=counter+1`. Figure 2.1.2 displays the use of a **det** block to resolve the explicit nondeterminism of a **choose** statement in a transition effects clause.

The use of **det** blocks is unaffected by the extension of the simulator to support composite automata simulations. Thus in our discussion of nondeterminism we concentrate on implicit nondeterminism. Above we saw two ways that implicit nondeterminism may arise during the simulation of a single automaton. Because at a given time during a simulation, more than one component may have enabled actions, simulating composite automata introduces a higher level of nondeterminism - which component should execute the next action? Once this nondeterminism, *high-level* nondeterminism, has been resolved and the next component has been identified, we once again face the two cases of nondeterminism above, which from now on will be referred to as *low-level* nondeterminism.

The user of the simulator has two options to resolve the implicit nondeterminism present in composite automata. The first option allows the user to choose a high-level nondeterminism resolution strategy and reuse the component schedule block to resolve low-level nondeterminism. The second option allows the user to write a composite schedule block and resolve both the high and low levels of nondeterminism at once.

2.1.2 Reuse of Component Schedule Blocks

This strategy to resolve nondeterminism present in composite simulations takes advantage of the fact that each component of the composition is itself an automaton that might have been simulated on its own. This implies that each component already has its own program to resolve the low-level nondeterminism. Thus all that is left to be done in order to simulate the composite automaton is the resolution of the high-level of nondeterminism. We provide the user with three options of how the high-level nondeterminism should be resolved during the simulation. These three options are similar to the three scheduling policies in Chefter's Scheduler [Che98]. However, there they refer to low-level nondeterminism and the selection of actions to execute. Here we extend this approach to high-level nondeterminism and the selection of the next component to contribute an action. Regardless of the high-level policy, it is possible that, once selected, a component will not be able to return an enabled action to the simulator. This situation is discussed in Section 2.1.3, after the three high-level policies are presented.

Strictly Uniform Policy

Chefter refers to this policy as round robin. Consider the simulation of a system with n components, $c_1, c_2, ..., c_n$. If the user selects the strictly uniform policy, then at the first step of the simulation the simulator will ask c_1 for the next action to execute (c_1 will use its own low-level nondeterminism resolution program to provide this action). At the second step, the simulator will ask c_2 for an action. This will continue until the $(n+1)^{th}$ step. At the $(n+1)^{th}$ step the simulator will once again poll c_1 for an action, and so on. The ordering of components is the same as their ordering in the IOA file in which their specifications appear. This is the default high-level policy.

Random Policy

This policy has the same name in Chefter's paper. If the user selects this policy, then at the beginning of each step the simulator randomly selects a component. The selection is entirely random and the component chosen at step k is completely independent of the components chosen at steps 1 through $(k-1)$. The presence of the boolean command line parameter **randComp** enables this policy in the simulator.

Weighted Policy

This policy is analogous to Chefter's policy that uses time estimates for each action. If the user selects this policy, then he/she must also provide a weight for each component. At each simulation step the simulator will pick component c_i with probability $\frac{\text{weight of } c_i}{\text{total weight of all components}}$. In this way this policy is similar to the random policy. The only difference is that here the components do not necessarily have an equal chance to get picked.

This policy is enabled via the command line parameter **weightComp**. This parameter is followed by one argument - the name of the file containing the weight specifications for the components. The syntax of the weight file is the following:

```
schedule
  states counter:Int:=1

do
  while true do
     if(Aut.total>10) then
        fire output action1
     fi;
     counter:=counter+1;
     if(Aut.ready=true) then
        fire output action2
     fi
  od
od
```

Figure 2.1.3: Example of a non-terminating NDR program

$$componentName_1 = integerWeight_1$$
$$componentName_2 = integerWeight_2$$
$$\ldots$$

The following checks are performed to ensure the validity of the specified weight file. If the file is found to be invalid, the simulation halts.

1. The file with the specified name exists,
2. Each line in the file contains exactly one, = symbol,
3. The text following the = symbol is a nonnegative integer, and
4. The text before the = symbol exactly matches a component name.

The user is not required to specify weights for all components. Those components whose weight is left unspecified will never be picked by the high-level strategy. If the user lists a particular component more than once, its weight will be the sum of the listings. The code changes necessary to support this strategy are described in Section 2.3.

2.1.3 Low-Level NDR Looping

Above we mentioned that when polled for an action the low-level nondeterminism resolution (NDR) program of the component might not find an enabled action. This can happen in two ways depending on whether the NDR program is terminating or not. Figure 2.1.1 and Figure 2.1.3 show examples of respectively a terminating and a non-terminating NDR program for automaton *Aut*. In either case, the simulator simply returns to the high-level NDR strategy and picks the next component to poll for an action. This is the next component in the ordering in the uniform case, and a randomly, possibly weighted, chosen component in the other two high-level NDR strategy cases.

If the component was unable to return an action because its low-level NDR block is terminating and is now exhausted, the simulator will not poll this component again during the simulation. However, notice that if a component with a non-terminating low-level NDR block is polled and it cannot return an action, the NDR block will loop infinitely. This will cause the simulator itself to loop infinitely. This is accepted behavior in the primitive automaton simulation case because if the NDR block of the single automaton cannot provide an action, no actions will be executed again and the simulation is in effect completed. This, however, is not the case in the simulation of a composite automaton. A component might have enabled actions at the point that another component's low-level NDR block has gone into an infinite loop. Below we first discuss two strategies to avoid this pitfall and then describe the implementation of the chosen one. A third strategy is presented as a future work in Section 8.3.

Maximum NDR Steps

One possible way to avoid the problem of a component's low-level NDR program looping forever, is to impose a limit on the number of steps that any NDR program may run for. Thus when a component is picked by the high-level NDR strategy, if its low-level NDR program runs for the alloted number of steps without returning an action, the simulator returns to the high-level NDR strategy and picks another component, as mentioned above. After this occurrence, the component is not eliminated from being picked again.

Taking Advantage of While Loop Structure

Another possible solution to the problem takes advantage of the fact that most non-terminating low-level NDR programs appear inside a `while(true)` loop. This solution proposes that when a low-level NDR program reaches the end of its `while(true)` loop, it relinquishes control back to the simulator. The simulator then uses the high-level NDR strategy to pick another component. Here also, the component is not eliminated from being picked again.

Implemented Strategy

The advantage of the Maximum NDR Steps strategy is that it does not rely on the structure of the low-level NDR program. Although, it is common for NDR programs to exhibit the mentioned structure, they are not required to do so. Thus if the second strategy were implemented, NDR programs that did not exhibit the `while(true)` loop structure would not avoid the pitfall of infinite looping. The advantage of the second strategy is that it does not have to pick a potentially arbitrary number as the limit on the low-level NDR steps allowed during a single iteration.

We made the decision to implement the first strategy, as the possibility of non standard structured low-level NDR programs causing infinite loops appeared particularly unpleasant. The user now has the option to specify the maximum number of steps that a low-level NDR program runs for during a single request for an action. This option is presented via the **ndrSteps** command line parameter. The parameter is defaulted at 500.

Note that the implementation of this strategy affects primitive simulations. We mentioned above that if a low-level NDR block loops in a primitive simulation, the entire simulation loops. With the introduction of the maximum steps limitation, instead of the simulation looping, it will halt and display an appropriate message. In the case of a primitive simulation with an unusually long NDR block, the user might want to increase the **ndrSteps** parameter to avoid the possibility of the simulation halting without the NDR block looping. The code changes necessary to support the **ndrSteps** parameter are described in Section 2.3

2.1.4 Composite Schedule Blocks

Above we discussed the possibility of reusing component schedule blocks to resolve nondeterminism in a composite automaton. It might be the case that the user does not want to reuse the existing component NDR blocks, but instead create a new NDR block tailored specifically for the composite automaton. In each **fire** invocation of this composite NDR block, the user specifies both the component that is executing the action and the action being executed. In doing so, the user resolves both the high-level nondeterminism, by specifying the component, and the low-level nondeterminism, by specifying the action.

The modifications to the IOA language required to support composite NDR blocks are discussed in Chapter 4. NDR blocks for composite automata are very similar to those for primitive automata. The two differences being:

- The action in each **fire** invocation of a composite schedule block must be prefixed with the component owning the action.

- References to component state variables of **all** components are allowed from within the composite schedule block.

Allowing the first modification requires component names to be of a sort that has access to all of the actions of the automaton that this component is based on. In IOA Toolkit release 0.09, read-only references to the state variables of the single primitive automaton are allowed from within that automaton's NDR block. These references are necessary because it is the responsibility of the schedule block to check precondition and where clause predicates before scheduling a transition. Allowing references to the state variables of all components, requires component names to be of a sort that has access to all of the state variables of the automaton that this component is based on.

2.2 Interaction of Components

In Section 1.2.1 we saw that the only way two components of a composite automaton may interact is through a same named action π. Moreover, the limitations posed on compositions, the sets of component internal actions have to be disjoint from all other component actions and each output action is controlled by a single automaton, limit this interaction to the execution of an output action π of one component causing the execution of input actions π of one ore more other components. This interaction also depends on the signatures of π matching and on the actual parameters of π satisfying the **where** clause of the input action.

Knowing that component interaction is limited to the above, the simulator must now simply execute all appropriate input actions π in the same step that the output action π was executed. Thus the only simulation steps during which an interaction is possible, are those steps that begin by executing an output action. When deciding how to implement this feature, two options were considered. The pros and cons of the options deal with the existence of **where** clauses in the transitions of π. These implementation options are discussed in detail in the following section.

2.2.1 Where Clauses in Transitions

One possible implementation of interaction between components involved building a map from the output transitions of components to zero or more input transitions of other components such that the execution of the output transition would cause the input transition(s) to be executed. This map would be built before any simulation steps took place. Thus every time an output transition would be executed during the simulation, the simulator would simply consult the map to see if any input transitions need to be executed as well. An alternative option had the map being built during the simulation. Here every time an output action were executed, the input actions corresponding to it would be populated in the map.

The problem with this implementation was discussed in Section 1.2.3, and involves the **where** clauses of the transitions of π. In order to build the map above, the simulator would have to find the intersections of the **where** clauses of the output and input transitions π. As we saw in table 1.2.1 this is often a difficult task, which at times (depending on the complexity of the predicate in the **where** clause) may be undecidable and require a proof. We thus decided to abandon the above implementation.

Instead, the simulator waits for the output transition π to be executed. Having done so, the simulator takes advantage of the fact that it has access to all of the actual parameters of π and using these parameters can evaluate the **where** clauses of all input actions π to see if they should also be executed. If an interaction does occur the simulator does not record the connection between the output transition π and the triggered input transition(s) π for future purposes. This is due to the fact that the next time output transition π is executed its actual parameters might not cause the same set of input transitions π to be executed.

Even with the implemented strategy above, a limitation on the type of predicates that may appear in the **where** clauses must be placed. Namely, the predicate must not contain either the existential or the universal quantifier. Just as discussed in Section 1.2.1 , the presence of such quantifiers would require a proof.

2.2.2 Scheduled Input Actions

An interesting situation arises when component schedule blocks are being reused to resolve non-determinism in a composite simulation, described in Section 2.1.2. What if input transition π of component A, that may be triggered by output transition π of component B, is scheduled in the low-level NDR block of component A? Two strategies were considered when dealing with such situations:

1. Look for and execute output transition(s) π when input transition π is executed,

```
automaton Aut1
   signature input  a
             output b
...

schedule
do
  while(true) do
    fire output b;
    fire input  a
  od
od

automaton Aut2
   signature input  b
             output c
...

schedule
do
  while(true) do
    fire output c;
    fire input  b
  od
od

automaton CompositeAut
   components Aut1;Aut2
```

Figure 2.2.4: Execution of scheduled input actions

2. Ignore all scheduled input transitions that at some point may be triggered by the execution of output transitions.

We made the decision to implement the second strategy. Since the output action is the driving force behind the interaction of components, it makes more sense for it to trigger the input action and not vice versa. An input action π that gets triggered by an output action in the composition, may be scheduled in the original component because when it is simulated as a stand alone automaton, the NDR block of this component mimics its external environment. Once this automaton is composed with another one (that has an output action π), and becomes a component in a more complex automaton, its schedule block no longer needs to mimic the external environment because the system becomes closed as a result of the composition. To find out whether a scheduled input action may eventually be triggered by an output action, we use the strategy described in Section 2.2.1 that was used to find out what input actions are triggered by the scheduled output action.

Figure 2.2.4 shows partial specifications of component automata *Aut1*, *Aut2*, and the composite automaton *CompositeAut*. We consider simulating *CompositeAut* and reusing the component schedule blocks to resolve nondeterminism. When the simulator encounters input action a in the schedule block for *Aut1*, it will execute it. The system represented by *CompositeAut* is not completely closed (*Aut2* does not have an output action named a), and the firing of input action a still mimics the external environment of *CompositeAut*. However, when the simulator encounters input action b in the schedule block for *Aut2*, it will not execute it. This action represents part of the system that has become closed due to the composition of *Aut1* and *Aut2*. It will only be executed as a result of output action b of component automaton *Aut1* being executed.

2.3 Code Changes

The class `simulator/shell/SimShell` has been modified to accept the **ndrSteps**, **randComp**, and **weightComp** parameters and to display them for simulator help. The value of the **ndrSteps** parameter is stored in the `simulator/Simulator` class. The class `simulator/ExecControl` has been modified to enable limited step execution. The class `simulator/StepsExceededProduct` has been created to represent the event of an NDR program exceeding the alloted number of steps.

The verification of the specified weight file provided with the **weightComp** parameter is done in `simulator/ActualCompositeAutomaton`. This class also builds the representation of the weight distribution. It contains an array that maps weight ranges to component indices and a hash table that maps the indices to the component names.

For more code change detail please consult Appendix B.

Chapter 3

Examples

This chapter contains five examples of composite automata that we would like to simulate. We break the examples down based on the choice of nondeterminism resolution strategy, reuse of component blocks versus use of a composite schedule block, and on the absence versus presence of component parameters. The first four examples cover the four combinations of the above scenarios. The fifth example illustrates the use of a **with** block within a composite schedule. The **with** block is a new IOA notion and is discussed in detail in Section 3.3. Chapter 4 discusses the IOA language extensions necessary to support the simulation of these examples. Chapter 5 discusses the simulator extensions necessary to support the simulation of these examples.

3.1 Examples with Non-Parameterized Components

The first two examples we consider are slight modifications of the toy banking system of [GL00]. The banking system consists of a single account that may be referenced from multiple locations. Automaton *Env*, Figure 3.1.2 , represents the outside environment of the banking system. The locations are indexed by the integer i. *Env* describes what operations can be invoked, where, and when. Notice that the only state kept by this automaton is a boolean flag for each location. This enables the environment automaton to request transactions at a certain location only once the previous transaction at that location has completed and *Env* has been informed of its completion. The actions of this automaton provide an interface for its communication with the *Bank* automaton.

Automaton *Bank*, Figure 3.1.3, is a mirror image of the *Env* automaton - output actions of *Env* are input actions of *Bank* and vice versa. Automaton *Bank* contains an additional internal action,

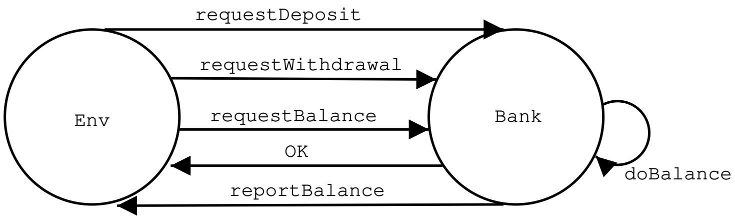

Figure 3.1.1: Actions of *Env* and *Bank*

```
automaton Env
  signature
      input     OK(i: Int,y:OpRec), reportBalance(n, i:Int)
      output    requestDeposit(n,i:Int) where n>0,
                requestWithdrawal(n,i:Int) where n>0, requestBalance(i:Int)
  states        active:Array[Int,Bool] :=constant(false)
  transitions
      input     OK(i,y)
                  eff active[i] :=false
      input     reportBalance(n,i)
                  eff active[i] :=false
      output    requestDeposit(n,i)
                  pre ¬active[i]
                  eff active[i] :=true
      output    requestWithdrawal(n,i)
                  pre ¬active[i]
                  eff active[i]:=true
      output    requestBalance(i)
                  pre ¬active[i]
                  eff active[i] :=true
```

Figure 3.1.2: Banking environment automaton

`doBalance`. As transaction requests are received by *Bank*, they are recorded. At some point after a request for the balance at a certain location is received, the `doBalance` transition calculates the balance at that particular location, and the balance at that location is now ready to be returned by `reportBalance`.

The interface diagram in Figure 3.1.1 shows that by matching external action of *Env* and *Bank*, we can form a new composite automaton *EnvBank*:

```
automaton EnvBank
  components Bank;Env
```

Automaton *EnvBank* is a combination of *Env* and *Bank*, and represents the bank composed with its environment. In order to be simulated in the presence of a simulator that is capable of handling only primitive automata, *EnvBank* would have to first be manually composed and then simulated. Test case `Banking01` of the IOA toolkit test suite does exactly this. We now see how automaton *EnvBank* is simulated directly.

```
automaton Bank
  signature
    input requestDeposit(n, i: Int) where n > 0,
          requestWithdrawal(n, i: Int) where n > 0,
          requestBalance(i: Int)
    output
          OK(i: Int,x:OpRec), reportBalance(n, i: Int)
    internal
          doBalance(i: Int, tempChosenOps: Set[OpRec], amount : Int)
    states ops: Set[OpRec] :={}, pending_ops: Set[OpRec] :={},
           reported_ops: Set[OpRec] :={}, bals: Set[BalRec] :={},
           pending_bals: Set[BalRec] := {}, done_bals : Set[BalRec],
           lastSeqno: Array[Int, Int] := constant(0),
           chosenOps: Set[OpRec]
  transitions
    input requestDeposit(n, i)
      eff  lastSeqno[i] := lastSeqno[i] + 1;
           ops := insert([i, lastSeqno[i], n, false], ops);
           pending_ops := insert ([i, lastSeqno[i], n, false], pending_ops);
    input requestWithdrawal(n, i)
      eff  lastSeqno[i] := lastSeqno[i] + 1;
           ops := insert([i, lastSeqno[i], -n, false], ops);
           pending_ops := insert ([i, lastSeqno[i], -n, false], pending_ops);
    input requestBalance(i)
      eff  pending_bals := insert ([i, 0], pending_bals);
           bals := pending_bals ∪ done_bals;
    output OK(i,x)
      pre   x ∈ ops ∧ x.loc = i ∧ ¬x.reported
      eff   ops:=insert(set_reported(x,true),delete(x,ops));
            pending_ops := delete(x,pending_ops);
            reported_ops:=insert(set_reported(x,true),reported_ops)
    output reportBalance(n,i)
      pre   [i,n] ∈ done_bals
      eff   done_bals :=delete([i,n],done_bals);
            bals:=pending_bals ∪ done_bals
    internal doBalance(i,tempChosenOps,amount)
      pre     [i,0] ∈ pending_bals
      eff     chosenOps:=tempChosenOps;
              pending_bals:=delete([i,0],pending_bals);
              done_bals :=insert([i,amount],done_bals);
              bals:=pending_bals ∪ done_bals
```

Figure 3.1.3: Automaton *Bank*

3.1.1 Example : Reuse of Component Schedule Blocks

Suppose, the component automata, *Env* and *Bank* had schedule blocks associated with them. Let the schedule block in Figure 3.1.4 be part of the specification of automaton *Env*, and the schedule block in Figure 3.1.5 part of the specification of automaton *Bank*. These automata can now be simulated as stand alone automata. Their output actions are simply not "heard" by anyone. The automaton *EnvBank*, closes the system. Notice that for the composite simulation, the *numLocations* and *maxAmount* schedule block variables in Figures 3.1.4 and 3.1.5 are set to the same value to avoid ambiguity.

The extended simulator can now simulate the composite automaton *EnvBank* directly. Assuming that the checker has been used to compile the `ioa` file containing the specification of *EnvBank* into the intermediate language file, `EnvBank01.il`[1], we can start the simulation by entering the following string at the command line:

```
sim 10 EnvBank EnvBank01.il
```

The default high-level nondeterminism resolution strategy is strictly uniform ordering. The command line invocation,

```
sim -randComp 10 EnvBank EnvBank01.il
```

would cause the random strategy to be used. At each step of the simulation, the next component is selected based on the high-level nondeterminism resolution strategy provided at the command line. The NDR block of this component is now executed until an action is fired or the NDR block exceeds the maximum number of steps alloted to it. If the fired action happens to be an output one, the input actions of all other components are checked for a possible triggering. The simulation now returns to its highest level of nondeterminism and the selection strategy is once again used to select an NDR block of a particular component to be executed.

[1]This can be accomplished by running, `ioaCheck -il EnvBank01.ioa > EnvBank01.il`, where EnvBank01.ioa is the file containing the specification of *EnvBank*

```
schedule
states
  numLocations, location, actionChosen, maxAmount : Int,
  op : OpRec, tempOps : Set[OpRec] := {}, tempOps2 : Set[OpRec] := {},
  tempBals : Set[BalRec] := {}, bal : BalRec, amount : Int,
  loopBreak : Bool := false
do
  numLocations := 15;
  maxAmount := 100;
  while (true) do % We'll pick a random location now
    location := randomInt (0, numLocations - 1);
    actionChosen := randomInt (0, 5);
    if (actionChosen ≥ 0 ∧ actionChosen ≤ 2) then
      % Do a deposit.  But must be sure we're not active at this location
      if ¬Env.active[location] then
        fire output requestDeposit(randomInt (1, maxAmount), location)
      fi
    fi;
    if (actionChosen ≥ 3 ∧ actionChosen ≤ 4) then
      if ¬Env.active[location] then
        fire output requestWithdrawal (randomInt (1, maxAmount), location)
      fi
    fi;
    if (actionChosen = 5) then
      if ¬Env.active[location] then
        fire output requestBalance (location)
      fi
    fi
  od
od
```

Figure 3.1.4: The NDR block for automaton *Env*

```
schedule
states
  numLocations, location, actionChosen, maxAmount : Int,
  op : OpRec, tempOps : Set[OpRec] := {}, tempOps2 : Set[OpRec] := {},
  tempBals : Set[BalRec] := {}, bal : BalRec, amount : Int,
  loopBreak : Bool := false
do
  numLocations := 15; maxAmount := 100;
  while (true) do % We'll pick a random location now
    location := randomInt (0, numLocations - 1);
    actionChosen := randomInt (0,4);
    if (actionChosen ≥ 0 ∧ actionChosen ≤ 2) then
      tempOps := Banking01.pending_ops; loopBreak := false;
      while (¬isEmpty(tempOps) ∧ ¬loopBreak) do
        op := chooseRandom (Banking01.ops);
        tempOps := delete (op, tempOps);
        if (¬op.reported) then loopBreak := true;
          fire output OK (op.loc, op)
        fi od fi;
    if (actionChosen = 3) then
      tempBals := Banking01.done_bals;
      loopBreak := false;
      if (¬isEmpty(tempBals)) then bal := chooseRandom (tempBals);
        tempBals := delete (bal, tempBals);
        fire output reportBalance (bal.value, bal.loc)
      fi fi;
    if (actionChosen = 4) then % Find a null balance
      tempBals := Banking01.pending_bals;
      loopBreak := false; bal := [10, 10];
      if (¬isEmpty(tempBals)) then
        bal := chooseRandom (tempBals);
        tempBals := delete (bal, tempBals);
        % There is a null bal to do balance for
        loopBreak := false; tempOps := Banking01.ops;
        tempOps2 := {};
        while (¬isEmpty(tempOps)) do
          op := chooseRandom(tempOps);
          tempOps := delete (op, tempOps);
          if (op.loc = bal.loc) then
            tempOps2 := insert (op, tempOps2)
          fi od;
        tempOps := tempOps2; amount := 0;
        while (¬isEmpty(tempOps)) do
          op := chooseRandom(tempOps);
          tempOps := delete (op, tempOps);
          amount := amount + op.amount
        od;
        fire internal doBalance (bal.loc, tempOps2, amount)
      fi fi od od
```

Figure 3.1.5: The NDR block for automaton *Bank*

3.1.2 Example : A Composite Schedule Block

Above we saw that when simulating a composite automaton, a user of the simulator may want to resolve nondeterminism by reusing the NDR blocks of the component automata. However, it may be the case that the user wants to resolve nondeterminism by writing a brand new NDR block for the composite automaton. In the former case, the component blocks serve the purpose of resolving the low-level nondeterminism while the command line selected strategy resolves the high-level nondeterminism. In this case, the composite NDR block resolves both levels of nondeterminism. Consider adding the NDR block in Figure 3.1.6 (declaration of schedule state variables has been ommitted for the purpose of brevity) to the specification of automaton *EnvBank*:

```
automaton EnvBank
  components Bank;Env
```

Notice that each **fire** invocation is now followed by the type of action being invoked as well as the component which owns the fired action, the action name, and the actual parameters, as in `fire output Bank.OK (op.loc, op)`. Similarly, there are references to the state variables of component automata, as in `Bank.done_bals`. The NDR block in Figure 3.1.6 was designed to closely model the NDR block used in the above mentioned test suite case `Banking01`, where the composite automaton was converted to a primitive one manually. Assuming that the checker has been used to compile the `ioa` file containing the specification of *EnvBank* and its composite schedule block into the intermediate language file, `EnvBank02.il`, we can start the simulation by entering the following string at the command line:

```
sim 10 EnvBank EnvBank02.il
```

Notice that this invocation does not differ from the one above where component schedule blocks were reused to resolve nondeterminism. The simulator uses the composite schedule block if it is present. If it is not, it defaults to reusing component schedule blocks. The above simulation produces the transition output displayed in Figure 3.1.7. When compared to the transition output of the manually composed automaton in the test case `Banking01`, Figure 3.1.8, we see that the two outputs are analogous.

```
schedule
do
  numLocations := randomInt(10,15); maxAmount := 100;
  while (true) do % We'll pick a random location now
    location := randomInt (0, numLocations - 1);
    actionChosen := randomInt (0, 10);
    if (actionChosen ≥ 0 ∧ actionChosen ≤ 2) then
      if ¬Env.active[location] then
        fire output Env.requestDeposit(randomInt (1, maxAmount), location)
      fi fi;
    if (actionChosen ≥ 3 ∧ actionChosen ≤ 4) then
      if ¬Env.active[location] then
        fire output Env.requestWithdrawal (randomInt (1, maxAmount), location)
      fi fi;
    if (actionChosen = 5) then
      if ¬Env.active[location] then
        fire output Env.requestBalance (location)
      fi fi;
    if (actionChosen ≥ 6 ∧ actionChosen ≤ 8) then
      tempOps := Bank.pending_ops; loopBreak := false;
      while (¬isEmpty(tempOps) ∧ ¬loopBreak) do
        op := chooseRandom (Bank.ops); tempOps := delete (op, tempOps);
        if (¬op.reported) then loopBreak := true;
          fire output Bank.OK (op.loc, op)
        fi od fi;
    if (actionChosen = 9) then
      tempBals := Bank.done_bals; loopBreak := false;
      if (¬isEmpty(tempBals)) then
        bal := chooseRandom (tempBals); tempBals := delete (bal, tempBals);
        fire output Bank.reportBalance (bal.value, bal.loc)
      fi fi;
    if (actionChosen = 10) then
      tempBals := Bank.pending_bals; loopBreak := false; bal := [10, 10];
      if (¬isEmpty(tempBals)) then
        bal := chooseRandom (tempBals); tempBals := delete (bal, tempBals);
        loopBreak := false; tempOps := Bank.ops; tempOps2 := {};
        while (¬isEmpty(tempOps)) do
          op := chooseRandom(tempOps); tempOps := delete (op, tempOps);
          if (op.loc = bal.loc) then
            tempOps2 := insert (op, tempOps2)
          fi od; tempOps := tempOps2; amount := 0;
        while (¬isEmpty(tempOps)) do
          op := chooseRandom(tempOps); tempOps := delete (op, tempOps);
          amount := amount + op.amount od;
        fire internal Bank.doBalance (bal.loc, tempOps2, amount)
      fi fi od od
```

Figure 3.1.6: Composite NDR block for automaton *EnvBank*

```
Automaton initialized
1:      output requestWithdrawal(9, 6) in automaton Env --- Connected to :
                input requestWithdrawal(9, 6) in automaton Bank
2:      output requestBalance(11) in automaton Env --- Connected to :
                input requestBalance(11) in automaton Bank
3:      output requestWithdrawal(74, 2) in automaton Env --- Connected to :
                input requestWithdrawal(74, 2) in automaton Bank
4:      output OK(2, [loc: 2, seqno: 1, amount: -74, reported: false])
                in automaton Bank --- Connected to :
                input OK(2, [loc: 2, seqno: 1, amount: -74, reported: false])
                in automaton Env
5:      output requestDeposit(36, 12) in automaton Env --- Connected to :
                input requestDeposit(36, 12) in automaton Bank
6:      output OK(12, [loc: 12, seqno: 1, amount: 36, reported: false])
                in automaton Bank --- Connected to :
                input OK(12, [loc: 12, seqno: 1, amount: 36, reported: false])
                in automaton Env
7:      internal doBalance(11, (), 0) in automaton Bank
8:      output requestWithdrawal(11, 9) in automaton Env --- Connected to :
                input requestWithdrawal(11, 9) in automaton Bank
9:      output OK(6, [loc: 6, seqno: 1, amount: -9, reported: false])
                in automaton Bank --- Connected to :
                input OK(6, [loc: 6, seqno: 1, amount: -9, reported: false])
                in automaton Env
10:     output requestWithdrawal(69, 1) in automaton Env --- Connected to :
                input requestWithdrawal(69, 1) in automaton Bank
No errors
```

Figure 3.1.7: Simulator output for composite automaton with a composite schedule block

```
Automaton initialized
1:      internal requestWithdrawal(9, 6) in automaton Banking01
2:      internal requestBalance(11) in automaton Banking01
3:      internal requestWithdrawal(74, 2) in automaton Banking01
4:      output OK(2, [loc: 2, seqno: 1, amount: -74, reported: false])
                in automaton Banking01
5:      internal requestDeposit(36, 12) in automaton Banking01
6:      output OK(12, [loc: 12, seqno: 1, amount: 36, reported: false])
                in automaton Banking01
7:      internal doBalance(11, (), 0) in automaton Banking01
8:      internal requestWithdrawal(11, 9) in automaton Banking01
9:      output OK(6, [loc: 6, seqno: 1, amount: -9, reported: false])
                in automaton Banking01
10:     internal requestWithdrawal(69, 1) in automaton Banking01
No errors
```

Figure 3.1.8: Simulator output for manually composed composite automaton

3.2 Examples with Parameterized Components

Before this project, automata parameters, both type and variable, were not supported by either primitive or composite simulations. This is not very surprising as automata parameters have little benefit to a simulator that can only handle primitive automata. Parameters allow for simple specifications of composite automata with multiple components based on the same automaton. For example the running example of the next few sections is a system that consists of multiple process and multiple channel components. Every process component is based on the same automaton. However, each one is instantiated with a different parameter. Variable parameters are now supported, while type parameters remain a fututre work.

The examples used in this and the following section, Figure 3.2.9, are slight modifications of the *Channel*, *Process*, and *Sys* automata used in the "Illustrative examples" section of Tauber's paper [Tau03]. The *Channel* automaton represents a communication channel that can drop duplicate messages and reorder messages. Notice the use of **const** parameters described in Section 1.1.2. The *Process* automaton represents a process that runs on a node indexed by the integer automaton parameter, n. This process communicates with its neighbors by sending and receiving messages that consist of natural numbers. The process records the smallest value it has received and passes on all values that exceed the recorded value; if the set of values waiting to be passed on grows too large, the process can also lose a nondeterministic set of those values [Tau03].

In the **components** definition of automaton *Sys*, *C* is a component name and *Channel* is a base automaton name. Component names appearing in the **components** specification will henceforth be referred to as component tags.

We now note that although the definition of automaton *Sys* in Figure 3.2.9 is a valid IOA specification, it is not sufficient for the purposes of simulating composite automata. The **where** clause presents the simulator with the problem of instantiating all of the components in its scope. Although it might seem that this is feasible in the case of automaton *Sys* in Figure 3.2.9, a more complex predicate involving `n` would force the simulator to search for all values satisfying the predicate. Thus without a theorem prover, it is not possible for the simulator to correctly instantiate all of the components scoped by **where** clauses. The interface diagram in Figure 3.2.10 shows the interaction of the *Process* and *Channel* components when all of the components in the scope of the two **where** clauses of Figure 3.2.9 are instantiated.

We first avoid this problem by considering two examples that involve specifications of the composite automaton *Sys* that do not contain a **where** clause. Later, we introduce the notion of a **with** block and illustrate its use through an example. A **with** block solves the scoping pitfall by requiring the user to instantiate all of the participating components.

```
automaton Channel(i, j:Int)
  signature
    input send(const i, const j, m:Int)
    output receive(const i, const j, m:Int)
  states contents:Set[Int] := {},
         formalI:Int:=i, formalJ:Int:=j
  transitions
    input send(i, j, m)
      eff contents := insert(m, contents)
    output receive(i, j, m)
      pre m ∈ contents
      eff contents := delete(m, contents)
automaton Process(n:Int)
  signature
    input receive(const n-1, const n, x:Int)
    output send(const n, const n+1, x:Int),
           overflow(const n, s:Set[Int])
  states
    val:Int := 0, toSend:Set[Int] := {}, formalN:Int:=n
  transitions
    input receive(n-1, n, x)
      eff if val = 0 then val := x
          elseif x < val then
            toSend := insert(val, toSend);
            val := x
          elseif val < x then
            toSend := insert(x, toSend) fi
    output send(n, n+1, x)
      pre x ∈ toSend
      eff toSend := delete(x, toSend)
    output overflow(n, s:Set[Int]; local t:Set[Int])
      pre s = toSend ∧ n < size(s) ∧ t ⊆ s
      eff toSend := t
automaton Sys
  components C[n:Int]: Channel(n, n+1) where 1 ≤ n ∧ n < 5;
             P[n:Int]: Process(n) where 1 ≤ n ∧ n ≤ 5
```

Figure 3.2.9: Automaton *Sys*

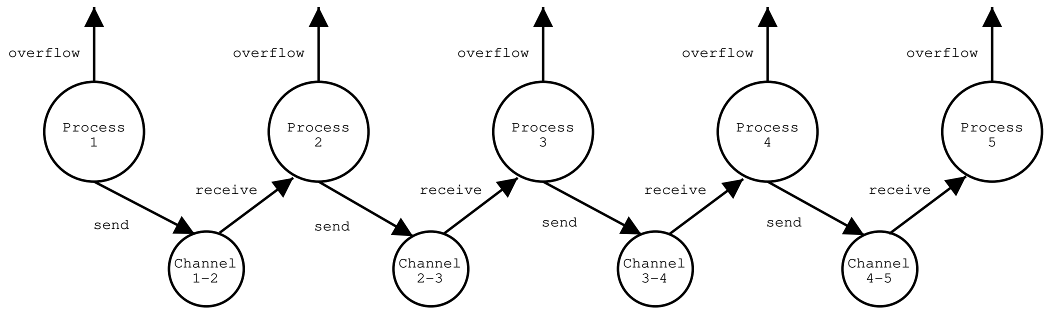

Figure 3.2.10: Components of automaton *Sys*

3.2.1 Example: Parameterized Components, Reuse of Component Schedule Blocks

Consider the following specification of automaton *Sys* as an alternative to the specification in Figure 3.2.9:

```
automaton Sys
  components C1: Channel(5,6);     P1: Process(5);
             C2: Channel(6,7);     P2: Process(6)
```

The automaton *Sys* now contains two components based on each base automaton. Since there is no composite schedule present, the simulation will reuse the component schedule blocks for *Channel*, Figure 3.2.11, and *Process*, Figure 3.2.12. We invoke the simulation with:

```
sim -outputTrans 8 Sys ProcChan01.il
```

Figure 3.2.13 shows the transition output of the simulation. Notice that the component tag names and not the base automaton names are used to identify the components in the output.

```
schedule
do
  while(true) do
    fire input  send(Channel.formalI,Channel.formalJ,290);
    fire output receive(Channel.formalI,Channel.formalJ,290)
  od od
```

Figure 3.2.11: NDR block for *Channel*

```
schedule
  states formVar:Int
do
  while(true) do
    formVar:=Process.formalN-1;
    fire input receive(formVar,Process.formalN,18);
    formVar:=Process.formalN+1;
    if(18 ∈ Process.toSend) then
      fire output send(Process.formalN,formVar,18)
    fi od od
```

Figure 3.2.12: NDR block for *Process*

```
Automaton initialized
1:      input receive(4, 5, 18) in automaton P1
2:      input send(6, 7, 290) in automaton C2
3:      input receive(5, 6, 18) in automaton P2
4:      input send(5, 6, 290) in automaton C1
5:      output send(5, 6, 18) in automaton P1 --- Connected to :
                input send(5, 6, 18) in automaton C1
6:      output receive(6, 7, 290) in automaton C2
7:      output send(6, 7, 18) in automaton P2 --- Connected to :
                input send(6, 7, 18) in automaton C2
8:      output receive(5, 6, 290) in automaton C1 --- Connected to :
                input receive(5, 6, 290) in automaton P2
No errors
```

Figure 3.2.13: Simulator output for automaton *Sys*, first case

3.2.2 Example: Parameterized Components, Composite Schedule Block

We now consider the case where the specification of automaton *Sys* remains as above:

```
automaton Sys
  components C1: Channel(5,6);    P1: Process(5);
             C2: Channel(6,7);    P2: Process(6)
```

However, now we provide it with a composite schedule block, Figure 3.2.14. Notice, that the references to the component state variables and component actions use the component tag names and not the base automata names. The invocation of this simulation, and its output are similar to those in the above section.

```
do
  while(true) do
    sendingAmount:=sendingAmount-1;
    amount:=P1.formalN-1;
    fire input P1.receive(amount,P1.formalN,sendingAmount);
    amount:=P1.formalN+1;
    fire output P1.send(P1.formalN,amount,sendingAmount);
    fire output C1.receive(C1.formalI,C1.formalJ,sendingAmount);
    amount:=P2.formalN+1;
    fire output P2.send(P2.formalN,amount,sendingAmount)
  od od
```

Figure 3.2.14: NDR block for automaton *Sys*

3.3 With Blocks

As mentioned above the simulator is not capable of instantiating components based solely on the **where** clause in the **components** declaration. The **with** block solves this problem by requiring the user to provide the simulator with all of the components that will be present in the simulation. Thus the burden of a possible proof is shifted from the simulator to the user. The **with** block is part of the schedule block and enumerates the components that will be created. Figure 3.3.15 shows an example of a **with** block that might appear in the schedule block of automaton *Sys* from Figure 3.2.9. We refer to each line in the **with** block as a *declaration.*

Declarations may only be made for component tags with formal parameters. Declarations for component tags with no formal parameters, would result in the composite automaton having more than one identical component, and this would violate the limitations on compositions established

```
schedule
  states
    randomInt:Int
  with
     compChannel1 = C[1],
     compChannel2 = C[2],
     compChannel3 = C[3],
     compP1 = P[1],
     compP2 = P[2],
     compP3 = P[3],
     compP4 = P[4]
do
     fire output compP2.send(1,2,10)
...
```

Figure 3.3.15: Example of a **with** block

in Section 1.2.1. Notice that in addition to enumerating the components that will participate in the simulation, the user provides a *handle* name for each instantiated component (the name on the left of the equals sign). As described below, the handle names are used in the schedule block of the composite automaton.

3.3.1 Handle Names in Schedule Blocks

Aside from the burden of a proof, **component** definitions with **where** clauses pose a difficulty for composite schedule blocks. In Section 2.1.4 we established that a **fire** invocation in a composite schedule block must be prefixed by the name of the component that is executing the action. One possibility for referencing the desired component in the schedule block is to list the component name followed by the actual parameters for that component. Such an invocation would look like:

```
fire output P[2].send(1,2,10)
```

The presence of the **with** block gives us the option to reference the component by its handle name. This is the implemented option and it looks like:

```
fire output compP2.send(1,2,10)
```

The benefit of the chosen option is brevity in the case of a component with many parameters. In such a case the user does not have to re-list all of the parameters every time he/she wants to reference the component. As mentioned in Section 2.1.4, the handle name can also be used in the schedule block to reference a state variable of a particular component.

In addition to their use in composite schedule blocks, handle names also provide an implementation benefit to the simulator itself. During a simulation of a composite automaton, the simulator must track all of the individual components. Handle names provide a convenient method for the simulator to uniquely identify all such components. Further, handle names are useful for output purposes in the simulation log file. To identify component variables and transitions, the simulator can display them prefixed by the component's handle name.

3.3.2 Non Exhaustive Declaration

We saw above how **with** blocks solve the problem of the necessity of a proof by enumerating the desired components. The simulator can now check the actual parameters provided in each of the declarations against the corresponding **where** clause. Just as in Section 1.2.1, we require that these **where** clauses do not contain quantifiers, otherwise the simulator would not be able to evaluate the **where** clause predicate even in the presence of actual values. If the parameter provided makes the **where** clause predicate evaluate to false, the simulator will halt the simulation and display an appropriate error message. Notice that although the simulator can detect illegal declarations, it can not verify whether or not the **with** block exhausts the **where** clause. Checking for an exhaustive declaration would require the same exact proof that motivated us to create the **with** block in the first place.

3.3.3 Example: With Block

Finally, we get to the most interesting case of a composite automaton with parameterized component automata. In this case the automaton *Sys* does contain a **where** clause in its **components** definition, which requires it to have a schedule block, and a **with** block in its schedule. Notice that the declarations in the **with** block happen to exhaust the **where** clauses. If this were not the case, the only effect on the simulation would be the presence of fewer components in the composite automaton. Automaton *Sys* is displayed in Figure 3.3.16.

The simulation begins with *Proc1* receiving a message "18". It should then send this message down the process-channel chain. The message should get to *Proc3* and stop there. Figure 3.3.17 displays the result of the simulation of composite automaton *Sys* for five steps.

```
automaton Sys
  components C[n:Int]: Channel(n, n+1) where 1 ≤ n ∧ n < 5;
             P[n:Int]: Process(n) where 1 ≤ n ∧ n ≤ 5
schedule
states
  amount:Int:=17
with
     Proc1=P[1],
     Chan12=C[1],
     Proc2=P[2],
     Chan23=C[2],
     Proc3=P[3],
     Chan34=C[3],
     Proc4=P[4],
     Chan45=C[4],
     Proc5=P[5]
do
  while(true) do
    amount:=amount+1;
    fire input Proc1.receive(0,1,amount);
    if(amount ∈ Proc1.toSend) then
       fire output Proc1.send(1,2,amount)
    fi;
    if(amount ∈ Chan12.contents) then
       fire output Chan12.receive(1,2,amount)
    fi;
    if(amount ∈ Proc2.toSend) then
       fire output Proc2.send(2,3,amount)
    fi;
    if(amount ∈ Chan23.contents) then
       fire output Chan23.receive(2,3,amount)
    fi od od
```

Figure 3.3.16: Automaton *Sys* case three : **with** block

```
Automaton initialized
1:      input receive(0, 1, 18) in automaton Proc1
2:      output send(1, 2, 18) in automaton Proc1 --- Connected to :
               input send(1, 2, 18) in automaton Chan12
3:      output receive(1, 2, 18) in automaton Chan12 --- Connected to :
               input receive(1, 2, 18) in automaton Proc2
4:      output send(2, 3, 18) in automaton Proc2 --- Connected to :
               input send(2, 3, 18) in automaton Chan23
5:      output receive(2, 3, 18) in automaton Chan23 --- Connected to :
               input receive(2, 3, 18) in automaton Proc3
No errors
```

Figure 3.3.17: Simulator output for *Sys* automaton that contains a **with** block

Chapter 4

IOA Language Extension

The ability to specify the automata and their constituents presented in the previous chapter required the IOA language to be extended to support:

1. NDR Blocks in Composite Automata,
2. Component Name Prefixes in Fire Invocations,
3. **with** blocks in composite schedule blocks, and
4. Declarations inside the **with** block.

The following four sections deal with the specific grammar changes involved with the above extensions, the new semantic checks necessary to verify the validity of the extended IOA files, the actual code modifications, and finally the intermediate language representations of the new extensions.

4.1 Grammar Modification

To allow NDR blocks in composite automata the grammar defining compositions was modified from,

```
composition      ::= 'components' component;+ ('hidden' actionSet)?
```

to:

```
composition      ::= 'components' component;+ ('hidden' actionSet,+)? compSchedule?
```

Now, a composition may or may not have a composite schedule block. The grammar dealing with composite schedule blocks is the following:

```
compSchedule        ::= 'schedule' states? withBlock? 'DO' compDetProgram 'OD'
compDetProgram      ::=  compDetStatement;+
compDetStatement    ::=   assignment | compDetConditional | compDetWhile | compDetFire
compDetConditional  ::=   'if' predicate 'then' compDetProgram
                              ('elseif' predicate 'then' compDetProgram)*
                              ('else' compDetProgram)? 'fi'
compDetWhile        ::= 'while' predicate 'do' compDetProgram 'od'
compDetFire         ::=  compInvocation1
compInvocation1     ::=  compInvocation2
compInvocation2     ::=  'fire' actionType compName '.' actionName invocationActuals?
                                                             ('case' IdOrNumeral)?
```

The structure of the grammar of a composite deterministic program, `compDetProgram`, closely resembles that of a primitive deterministic program. The exception is that a composite invocation is required to have a component name prefix. An alternative modification of the grammar had compositions being modified to the following:

```
composition      ::= 'components' component;+ ('hidden' actionSet,+)? schedule?
```

This strategy does not make a distinction between a schedule block for a primitive automaton and a schedule block for a composite automaton. Because of this, implementing this strategy would avoid the creation of new non-terminal symbols `compDetProgram`, `compDetStatement`, and `compDetFire`. However, a fire statement would now have to branch to both non-prefixed and prefixed invocations. This would require new semantic checks to ensure that no prefixes were specified in the invocations of NDR blocks of primitive automata. Thus although the chosen implementation involves the creation of more new symbols, it is more straightforward and cleaner.

The composite schedule block may or may not have a **with** block. We define the grammar of a **with** block and the declarations inside it to be:

```
withBlock        ::= 'with' declarations
declarations     ::=  declaration,+
declaration      ::=  handle EQ IDENTIFIER '[' terms,+ ']'
handle           ::=  componentName
```

Notice that the first member of a declaration is a `handle` and not an identifier like the member on the other side of the equals sign. This is necessary because the handle will be used in the schedule block to reference state variables and transitions of the component. Thus it must be a variable of type `componentNode` and not simply an `ltoken`. However, the identifier simply links the component being declared here to a component tag in the **components** definition, and it is sufficient to represent the identifier by an `ltoken`.

A possibility for future work would allow a more complicated code structure inside the **with** block. For example, declaring components inside a **for** loop might prove to be useful. A brief guide on modifying the IOA Grammar appears in appendix C.

4.2 Semantic Checks

The introduction of component name prefixes in **fire** statements presents one new semantic check. The prefix specified in the invocation must exactly match one of the component tags without formal parameters in the **components** declaration of the composition, or the handle name of one of the declarations of the **with** block. This semantic check is performed along with the rest of the semantic checks for a composite automaton. Semantic checks that confirm the existence of the specified action and match the validity of the type of the action as well as of the parameters specified, already exist. These checks are invoked on the owning component once the new semantic check has verified the existence of the specified component.

The component prefix should be a variable of sort that is an aggregate over all of the transitions of the component automaton. Its `.` operator should allow access to all of the transitions of the component automaton. Currently, the prefix is not implemented in this way. It is simply an `ltoken` that is easily parsed and verified because a component name is the only thing that may precede an action name in an invocation.

The following is a list of the semantic checks that must be performed to ensure the validity of the IOA code in a **with** block:

1. The handle names in the declarations must be unique.

2. Handle names must be distinct from the names of the state variables of the schedule block.

3. Handle names must be distinct from the formal variables of the composite automaton.

4. The indentifier on the right hand side of the equals sign must exactly match one of the component names in the **components** definition.

5. The length of the list of terms following the indentifier must equal the length of the list of formal parameters of the matching component from the **components** definition.

6. The types of the terms following the identifier must match the types of the formal terms of the matching component from the **components** definition.

7. Each of the terms following the identifier must be a simple literal term.

The first three semantic checks ensure that variable names remain unambiguous inside the composite schedule block. The fourth semantic check requires that each declaration correspond to some component tag established in the **components** section. The fifth and sixth checks guarantee that the terms in the declaration are valid with respect to the terms declared for the component tag in the **components** section. Finally, the seventh check puts a restriction on the type of terms that may appear here. Just as above, these semantic checks are performed along with the rest of the semantic checks for a composite automaton.

4.3 Code Changes

Table 4.3.1 lists the Java classes in directory (*IOA_Toolkit/Code/ioa/parser*) (we will refer to instances of these classes as parser side objects) that represent the new non-terminal symbols introduced to the grammar in Section 4.1. The class `compDetFireNode` is a new class that extends the existing `detFireNode` and provides the `set` methods used to create the composite invocation during parsing. The class `invocationNode` now has a field representing the component name that may be associated with the invocation statement. Similarly, `compositionNode`, the class that represents a composition, now has a field that represents the schedule block that may be associated with the composite automaton.

The class `withNode` is a new class and represents a **with** block. Its state consists of a collection of declarations. It has a method that retrieves a component tag name based on its handle name. The class `declarationNode` is a new class and represents a single declaration in a **with** block. Its state consists of a handle name, a component tag name, and a list of terms corresponding to the formals of this component tag. The class `componentNode` is an existing class. When it represents

non-terminal symbol	**Java class**
compSchedule	detScheduleNode
compDetProgram	ListNode of statementNode
compDetStatement	statementNode
compDetFire	detFireNode
compInvocation1	invocationNode
compInvocation2	compDetFireNode
withBlock	withNode
declarations	ListNode of declarationNode
declaration	declarationNode
handle	componentNode

Table 4.3.1: Java classes representing new non-terminal symbols

a `handle`, it does not have the correct state upon creation. Its state variables get updated later when this component is linked to the component declared in the **components** specification by the component tag name. For a more detailed documentation of the modified files please consult Appendix A.

4.4 IL Representations

The classes listed in Figure 4.3.1 all implement the method `makeAbstract`. This method converts the parser side object representing an automaton constituent into its counterpart object that is an instance of a class in the (*IOA_Toolkit/Code/ioa/automaton*) directory (we will refer to instances of these classes as automaton side objects). These automaton side, counterpart objects are all capable of translating their representation into intermediate language code. After all semantic checks have been performed, the `makeAbstract` method is invoked on all of the participating parser side objects and the automaton side objects are created.

The counterparts of `invocationNode` and `compositionNode` on the automaton side are `ndrfire` and `composition` respectively. These classes have been modified to account for the possibility of NDR blocks in composite automata and component name prefixes in the **fire** statements of those blocks.

The IL representation of an ndr **fire** statement has been modified from:

```
(FIRE {transition-id (ACTUALS actuals+)}?)
```

to,

```
(FIRE {component-name? transition-id (ACTUALS actuals+)}?)
```

where the capitalized words denote literal strings and lower case words denote IOA notions; curly brackets denote grouping and do not actually appear in the IL syntax. The IL representation of a composition has been modified from:

```
((COMPOSE {(component-name (ACTUALS actuals+))}+) (HIDDEN hiddens+))
```

to,

```
((COMPOSE {(component-name (ACTUALS actuals+))}+) (HIDDEN hiddens+) schedule?)
```

Compared to the IL representation of a a primitive schedule block,

```
(SCHEDULE (STATES states*) program)
```

the IL representation of a composite schedule block is,

```
(SCHEDULE (STATES states*) (WITH declarations+)? program)
```

and the IL representation of a declarations is,

```
(handle componentTag actuals+)
```

where the capitalized words denote literal strings and lower case words denote IOA notions.

Chapter 5

Simulator Extension

In Chapter 4 we saw how the IOA language and the tools that parse it have been modified to account for the possibility of composite schedule blocks, component prefixes in **fire** invocations, and **with** blocks. We now concentrate on the modifications to the simulator itself necessary to utilize these notions in order to support simulations of composite automata. We divide this chapter based on the presence of parameters in the components of the composite automaton and on the nondeterminism resolution strategy used to resolve nondeterminism in the composite automaton.

First, we describe the structure of the simulator classes that represent a composite automaton and its components. Second, we discuss how this structure is employed to allow more than one component to be based on the same automaton during simulations of a composite automaton with parameterized components. Next, we describe the connection via the IL parser of the intermediate language representation of an automaton to its representation in the simulator. Finally, we describe the modifications to the output produced by the simulator that are motivated by simulations of composite automata.

5.1 Representation of a Composite Automaton

The Java class representation of automata consists of two sides, the basic side and the actual side. The basic side is created during the parsing of the IL file. It is a representation of the blueprint of the automaton. The actual side is created at the beginning of the actual simulation, and allows for the addition of actual parameters to the automaton. Prior to this project, the representations of composite automata on both sides were either limited or nonexistent.

5.1.1 Basic Side

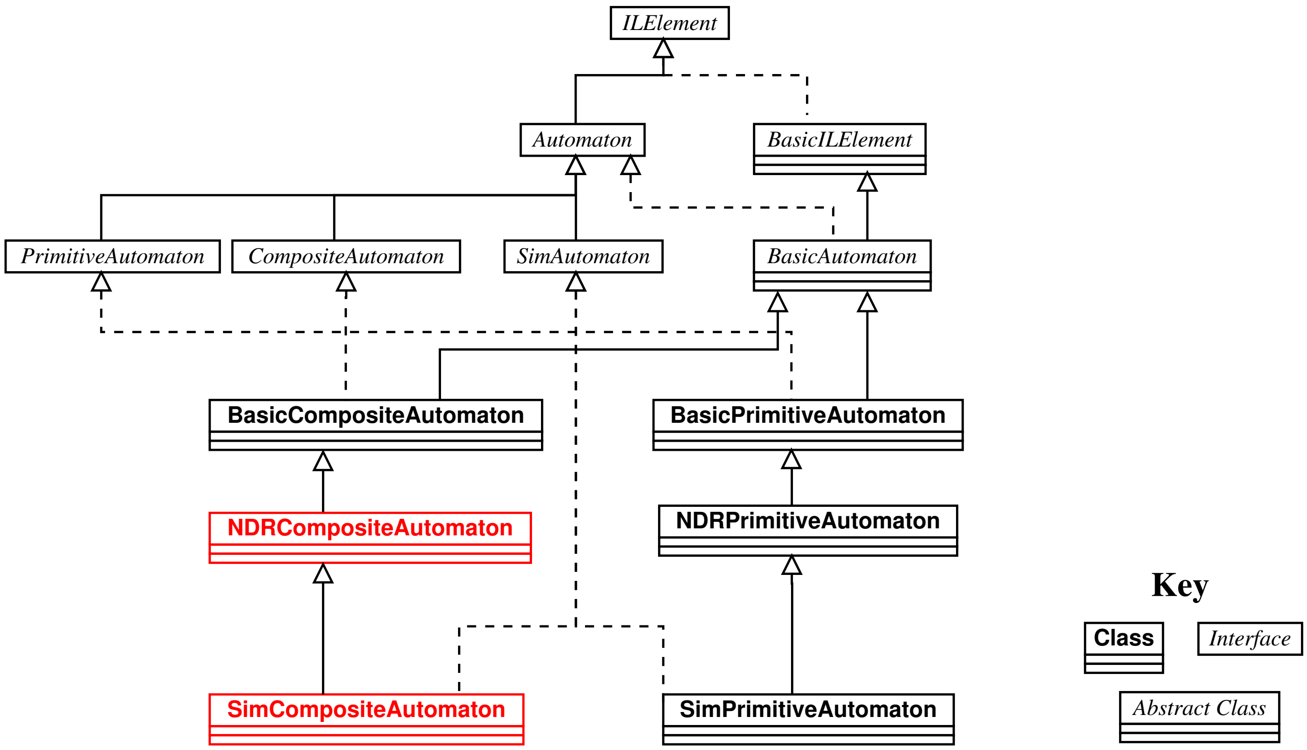

Figure 5.1.1: Basic side object diagram

Figure 5.1.1 shows part of the basic side architecture of the simulator. The components of a composite automaton are stored as a `Vector` of `AutComponent` objects in `BasicCompositeAutomaton`. The `NDRCompositeAutomaton` and the `SimCompositeAutomaton` objects have been added to the architecture. The `NDRCompositeAutomaton` object supports a schedule block in a composite automaton. The `SimCompositeAutomaton` object allows for instantiation of actual side objects based on their basic side representations. The method that does this is now declared in the `SimAutomaton` interface, instead of just in the `SimPrimitiveAutomaton`. Both `NDRCompositeAutomaton` and `SimCompositeAutomaton` objects parallel their primitive counterparts.

5.1.2 Actual Side

Just as the `SimAutomaton` interface is an abstraction for a basic side automaton, primitive or composite, the new interface `ActualAutInterface` is an abstract connection to an actual automaton, whether primitive or composite. Figure 5.1.2 shows the relevant part of the actual side architecture before the latest modifications. Figure 5.1.3 displays the architecture after them.

During the simulation, the `Simulator` object has access only to the new `ActualAutInterface` interface and is not aware whether it is simulating a primitive or a composite automaton. The `ActualCompositeAutomaton` represents a composite automaton, possibly with parameters. The actual components of the composite automaton are stored as a `Vector` of `ActualAutomaton` objects. Note that because simulations of composite automata whose components are themselves composite automata are not currently supported, this is sufficient. To support such simulations slight modifications would have to be made, including the storing of actual components as `ActualAutInterface` objects and not primitive `ActualAutomaton` objects.

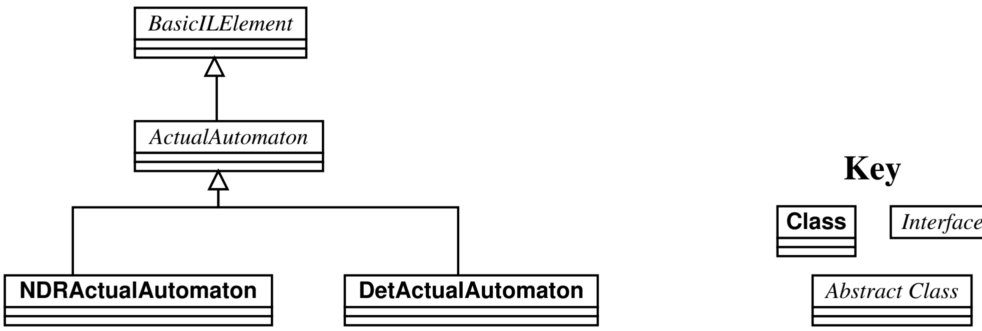

Figure 5.1.2: Actual side object diagram before changes

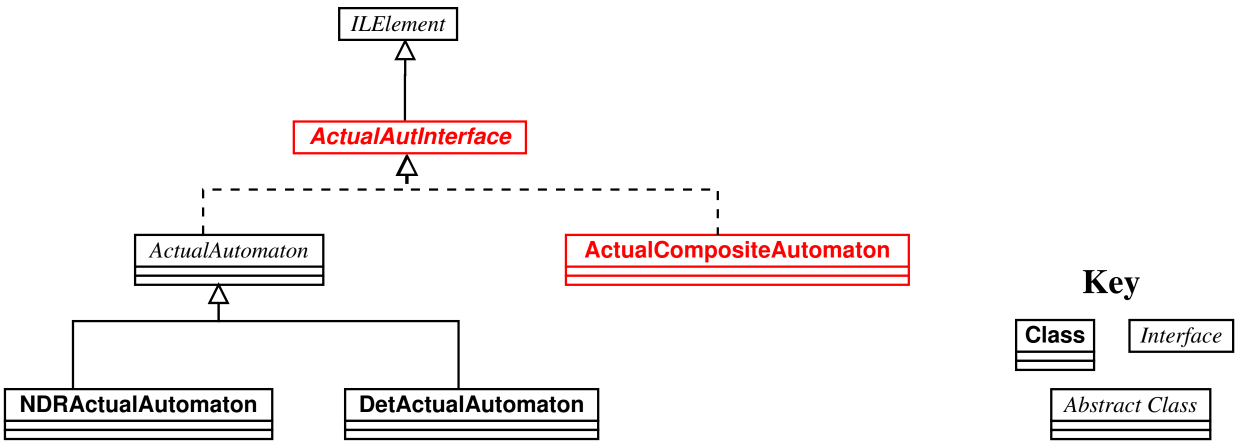

Figure 5.1.3: Actual side object diagram after changes

5.1.3 Implementation Decisions

An alternative to the above implementation of composite automata, was to create objects for composite automata that are not related to the primitive automata objects. In addition to this, create a new simulator object that specifically simulates composite automata. The biggest pro of this approach is the ease with which a primitive/composite specific change could be implemented. The major con of this approach is the double implementation and upkeep of features that are common to both primitive and composite simulations.

The approach chosen, avoids duplicate code upkeep. We took care to avoid large code blocks specific to primitive/composite simulations in the Simulator object. When necessary, the distinctions are handled through different implementations in the primitive and composite objects of the methods available to the Simulator through the interface .

For a more detailed documentation of the modified files discussed above please consult Appendix B.

5.2 Copying of Components

When the simulator tool was initially created, its implementation was tailored for the simulation of a primitive automaton. The possibility of simulating a composite automaton with parameterized components presents the problem of creating more than one simulator representation of an automaton based on a single IOA automaton specification. For example, when simulating automaton *Sys* from Figure 3.3.16, the simulator would have to create and track four components based on the *Channel* automaton. Initially, the only difference between these components are the values of the formal parameters of the automaton *Channel*. However, as the simulation progresses, the state of each of these components will diverge. This necessitates the creation of a separate copy of each component.

5.2.1 Implementation Decisions

Three reasonable options for creating more than one simulator representation of an automaton based on a single IOA automaton specification existed:

1. Augment the IOA parser to create multiple IL representations of a primitive automaton whenever the specification of a composite automaton with parameterized components is encountered.

2. Modify the IL Parser to parse the same IL automaton representation multiple times whenever the IL representation of a composite automaton with parameterized components is encountered.

3. Modify the Simulator, such that prior to the simulation, it creates copies of its representation of an automaton, whenever the simulation about to take place involves a composite automaton with parameterized components.

The decisive disadvantage of option one is its violation of the one-to-one correspondence between notions in an IOA file and their IL representations. The feasibility of option two depended on the current implementation of the IL Parser and the magnitude of the modifications to it that would produce the desired result. Currently, the IL Parser scans the IL file in a top-down manner. For example, when the parser encounters the **automaton** keyword, it expects a name, a list of formals, a list of actions, a list of states, and a list of transitions to follow immediately after.

In order to achieve the goal proposed by solution two, once the IL parser came across the definition of a composite automaton with parameterized components, it would have to return to the position in the IL file where the base automaton for that component were located, and then parse it top-down again. Implementing the ability of the IL parser to traverse the IL file backwards, in search for the base automaton, would require extremely significant modifications to it. Due to this, option three was chosen.

5.2.2 Code Changes

The implemented option causes the simulator, prior to the simulation, to create copies of its representation of an automaton, whenever the simulation about to take place involves a composite automaton with parameterized components. Figure 5.2.4 shows the object representation of a primitive automaton. Solid lines represent containment (with an asterisk denoting the containment of multiple objects), dotted lines represent read-only access, and dashed lines represent read/write access.

As can be seen from the diagram, a primitive automaton contains a collection of transitions whose `where` term, `precondition` term, and `effect` program can all access the state variables and formals of the automaton. Similarly, the NDR program of the automaton can access both the NDR variables and the state variables of the automaton. This scenario presents two options as to how the automaton may be copied.

One possibility is to copy the entire automaton representation hierarchy. After the copy is complete, a scan of the transitions is necessary to make them access the new copy of the state variables and not the original one. Since the only difference between transitions of every component based

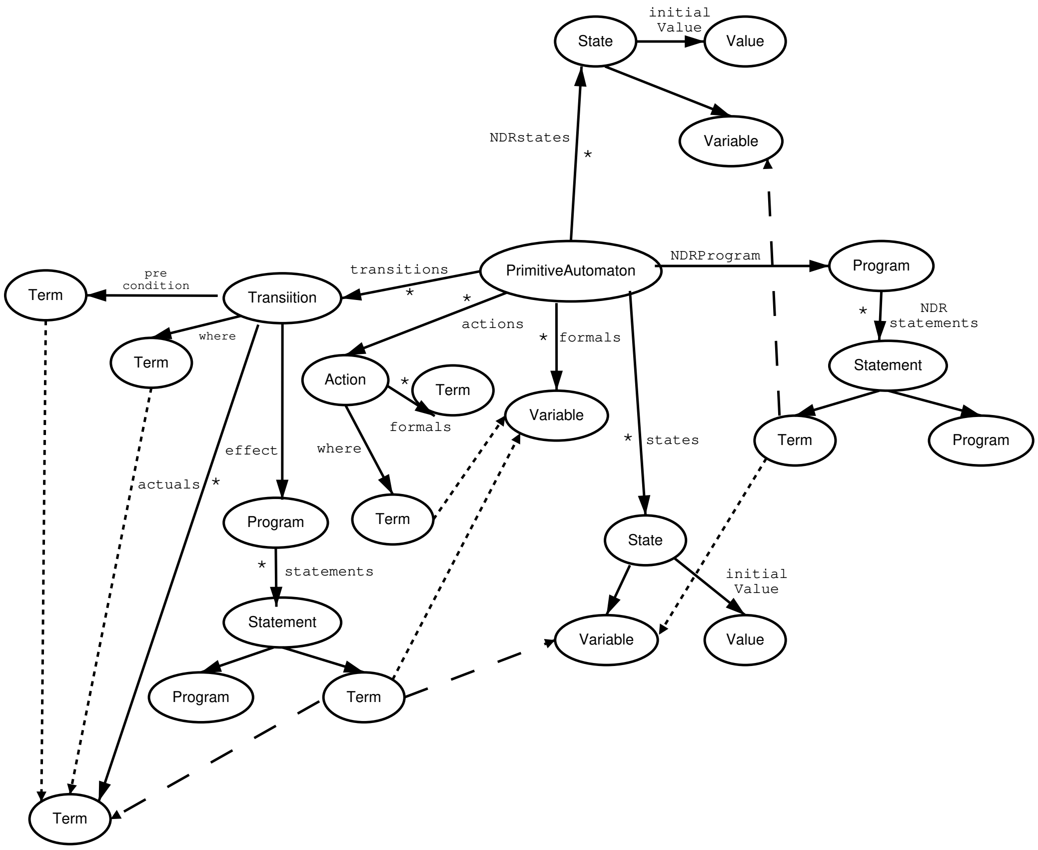

Figure 5.2.4: Partial, abstract representation of an automaton

on the same automaton, are the references to the objects representing state or formal variables, another option is to copy the state and formal variables only. A single set of transitions would be kept for all components based on the same automaton. Prior to the execution of a transition, its references to variables would get modified to point to those of the currently active component.

The advantage of the second implementation is that it does not copy objects that do not necessarily need to get copied. Its disadvantage is the necessity to update transition references every time a transition is processed. We decided that the first implementation is superior. The significant advantage of the first implementation is that after the copy and update are complete, the simulation can continue without needing to be interrupted again for copy/update purposes. This allows future modifications to the mechanism of the simulation to be independent of the copy/update process.

The situation is a bit different in the case of the NDR program and its variables. NDR programs differ from the **effect** programs in transitions because every time the *effect* program is ran, it runs until completion, and if its ran again, execution starts from the top. This is not the case with NDR programs. Their execution is interrupted by **fire** statements and resumes from the same spot on its next iteration. Thus NDR programs must either be copied, or a new method to maintain each components' place within its NDR program must be implemented. For this reason, NDR programs are also copied in their entirety and their variable references are updated just as is the case with transitions.

The copying of formals is fairly straightforward as they are shallow objects; their state consists of `String`'s, `boolean`'s and an `Entity` reference. The copying of both automata and NDR state variables, involves copying their initial value terms. The copying and updating of statements in `effect` and NDR programs is more complicated because they are objects with fairly deep state. To copy all of the above, we must be able to copy the terms that appear as initial values, `preconditions` and `where` clauses of transitions and the statements that appear in programs. Figure 5.2.5 shows an abstraction of such terms.

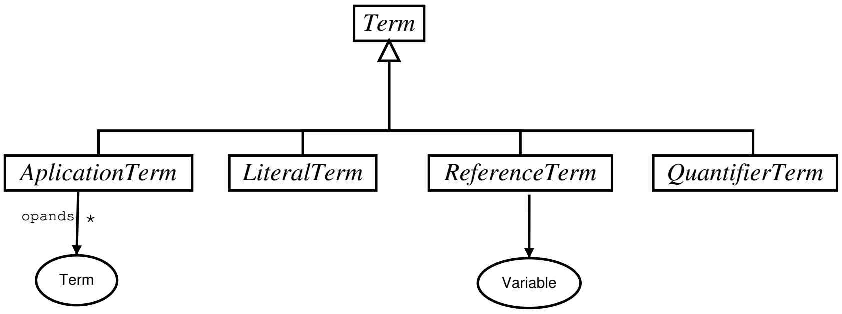

Figure 5.2.5: Partial, abstract representation of a term

Since we have excluded the possibility of quantifiers in **where** clauses in Section 2.2.1, we do not worry about quantifier terms here. Literal terms do not contain any dynamic state and do not have to be copied. Thus the only work remaining is the copying of the opands of application terms and updating the variable reference of reference terms.

Figure 5.2.6 shows an abstraction of statements. It shows what objects need to be copied and/or updated for each type of statement. The `Term`'s in the diagram are copied, the `Variable`'s are updated to point to the correct copy, and the `Program`'s are a collection of `Statement`'s that recursively get processed in the same way.

For a more detailed documentation of the modified files discussed above please consult Appendix B.

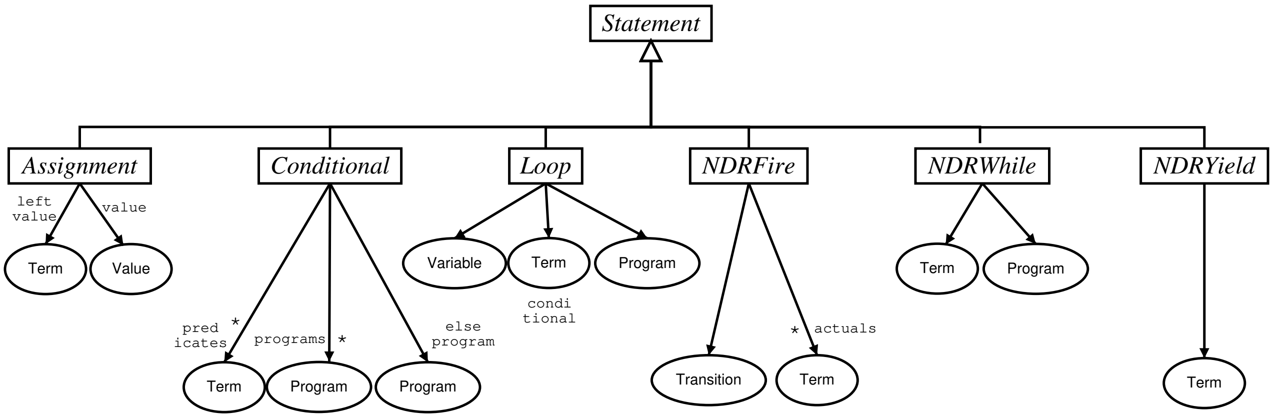

Figure 5.2.6: Partial, abstract representation of a statement

5.3 IL Parser

We saw in Section 4.4 that the intermediate language representation of composite automata and their constituents generated by the IOA parser has been modified. The back-end tool that in turn parses this code and creates simulator side objects discussed above is the IL parser, (*IOA_Toolkit/Code/ioa/il/ILParser.java*). The IL parser has been modified to account for the changes in the IL language.

Most of the modifications occur in the `parseCompositeAut` method. The parser now checks for the existence of a **with** block in a composite automaton. If it exists the parser creates two maps, one from component tag names to base automaton names, and another from handle names to component tag names. These maps are used by the `NDRILFactory` and `NDRCompositeAutomaton` classes when parsing the schedule block of the composite automaton. Finally, the parser creates as many component objects as are declared in the **with** block. These component objects are later used by the simulator to create copies of automata objects as described in Section 5.2.2. Also, as discussed in Section 5.4, when creating simulator representations of state variables, it lets the variable know which component it belongs to. For a more detailed documentation of the modified files discussed above please consult Appendix B.

5.4 Display of Output

The new functionality of the simulator discussed in the sections above requires some modifications to the way that its output is displayed. It is no longer sufficient to list modified state variables by their name alone as that would not make it clear to which component they belong. Also because an output action may trigger one or more input actions, it is now possible for more than one transition to be executed at every step of the simulation.

To solve the first problem the simulator representations of state variables have been modified so that they are now aware to which component they belong to. In composite simulations, every time a variable is displayed, the name of the owning component of that variable is displayed in front of it. Figure 5.4.7 illustrates this via a hypothetical output during the simulation of the *EnvBank* automaton described in Section 3.1. To solve the second problem, the output transition that initially triggers the input transitions is aware of the transitions triggered by it and it displays this information following its own display. Figure 5.4.8 illustrates this.

```
%%%% Modified state variables:
%%    Bank:ops --> ([loc: 8, seqno: 1, amount: 78, reported: false])
%%    Bank:pending_ops --> ([loc: 8, seqno: 1, amount: 78, reported: false])
%%    Bank:lastSeqno --> (ArraySort (ConstantValue 0) (8 1))
%%    Env:active --> (ArraySort (ConstantValue false) (8 true))
```

Figure 5.4.7: Display of state in composite simulations

```
[[[[ Begin step 1 [[[[
    transition: output requestDeposit(78, 8) in automaton Env --- Connected to :
                input requestDeposit(78, 8) in automaton Bank
```

Figure 5.4.8: Display of triggered transitions in composite simulations

Chapter 6

Test Suite Extension

The IOA toolkit contains a comprehensive test suite for regression testing purposes. The test suite contains tests for every tool in the kit. For our purposes, we are most interested in the `checker` and the `sim` tests that check the behavior of the IOA parser and the simulator respectively. The test suite consists of about one hundred test cases each designed to examine specific functions of the toolkit. The following sections deal with the test cases added to check the functionality of the tools for parsing IOA specifications of composite automata, the simulation of these automata, and with a parameter added to enable the automation of such tests. Because of the extra complexity involved in simulating composite automata with parameterized components, we separate the tests into those that do check such simulations, and those that do not.

6.1 SIMAUTOMATON Parameter

The test suite allows the user to run all tests on all of the test cases at once. The user also has the option to run a particular test on all of the test cases at once. The `Makefile` in the directory of each test case contains parameters that allow the test case to be customized. For example, the SIMSTEPS parameter specifies the number of steps that the test simulation should be run for and the SIMDEBUG parameter allows the user to specify the random seed to be used for the simulation.

Before the introduction of composite automata simulations, each IOA file in the test suite contained the definition of only one automaton. The possibility of composite simulations introduces the situation where an IOA file might contain more than one automaton definition. At the simulation command line, the simulator must then be provided with the name of the automaton to simulate. To allow for the automated testing of composite automata simulations, the `Makefile` of each test case now contains a SIMAUTOMATON parameter. When provided, this parameter specifies the name of the automaton to be simulated. Note that the introduction of this parameter does not require the modification of all existing `Makefile`'s because when this parameter is not specified, the simulator assumes that there is only one automaton in the IOA file and simulates it.

6.2 Non-Parameterized Components

The test cases described in this section involve composite automata with non-parameterized components. They test the simulator's behavior when using both the component schedule block reuse and composite schedule block strategies to resolve nondeterminism. Many of the scenarios checked in these tests also apply to the simulations of composite automata with parameterized components. Section 6.3 describes the test cases for such simulations. Those tests build on the ones described in this section by concentrating on issues particular to simulations of composite automata with parameterized components.

6.2.1 Testing Reuse of NDR Blocks

The following tests check the behavior of the simulation of composite automata which do not contain a composite schedule block and therefore use the NDR blocks of the components to resolve nondeterminism. We first must test that if a component name prefix is specified in a NDR block of a primitive automaton, the IOA checker displays an error message. The following is a list of points to be checked on the simulator side. Note that the first five cases are independent of nondeterminism resolution strategy and thus also apply to the tests in Section 6.2.2.

1. All input actions π, whose **where** clause is satisfied by the actual parameters, are executed in the same step that output action π of some other component is executed.

2. No input actions other than π, whose **where** clause is satisfied by the actual parameters, are executed in the same step that output action π of some other component is executed.

3. Correct behavior of the above cases for actions with and without actual parameters.

4. Component invariants are verified correctly.

5. Composite invariants are verfied correctly.

6. Input action π that is scheduled to be fired by its component's NDR block, but may also be triggered by the execution of output action π of some other component, does not get executed when its NDR block attempts to fire it.

7. Component NDR blocks that are terminating/non-terminating behave correctly.

8. Component NDR blocks that at some point become infinitely looping are correctly disrupted by the NDR max steps parameter.

9. High-level nondeterminism resolution strategies - uniform, random, and weighted all behave as expected.

The following is a list of newly created test cases that have been added to the test suite. If left unspecified, the high-level nondeterminism resolution strategy is uniform.

- `ComposedBank01` test case. This test case corresponds to the automaton discussed in Section 3.1.2. The composite automaton contains a bank component and an environment component. This test is similar to the `Banking01` test case, except here we have a direct simulation of a composite automaton, while there the composite banking automaton is created manually. The SIMAUTOMATON parameter for this test case is set to *EnvBank01*, the name of the composite automaton. The random seed, `rseed`, value is set to 10 just as in the `Banking01` test case. This test checks points 1 and 7 above.

- `PushPullAut01` test case. The two components of the composite automaton *PushPullAut* are *PushAut* and *PullAut*. Their respective input/output actions are connected to each other. The pre-conditions on the output actions are such that they require the two components to fire their output actions alternatively. The component *PushAut*, also has an input action that is connected to *PullAut*. The SIMAUTOMATON parameter for this test case is set to *PushPullAut*. This test checks points 3, 5, 7, and 8.

- `PushPullAut02` test case. Here the composite automaton, *PushPullAut* contains three components. The output action of *PushAut* is connected to input actions of both *PullAut* and *ExtraAut*. The output action of *PullAut* is only connected to an input action of *PushAut*. The component *ExtraAut* contains two output actions that do not trigger any input actions. The SIMUATOMATON parameter for this test case is set to *PushPullAut*. This test checks points 2, 4, 5, 8, and 9.

There is a difficulty involved with automated testing of the random and weighted high-level nondeterminism resolution strategies. Because the components are selected based on a random number picked by the Java random number generator, there is no way to make two consecutive simulations produce the same output. Thus although these strategies have been tested, there are no test cases in the automated test suite that check the simulator's behavior under these high-level nondeterminism resolution strategies.

6.2.2 Testing NDR Blocks for Composite Automata

The following test checks the behavior of the simulation of a composite automaton which contains a composite schedule block. We must first check that the IOA parser handles such automata correctly. The following is a list of cases to be verified on the IOA parser side.

1. Composite schedule blocks are accepted.
2. Component name prefixes in composite **fire** invocations are accepted.
3. Only appropriate component name prefixes in composite **fire** invocations are accepted (Semantic Check).
4. Component name prefixes are required in composite **fire** invocations.
5. Actions invalid for the specified component in a composite **fire** invocation are discovered and reported.

As mentioned in Section 6.2.1, the first five simulator test points listed there also apply to testing composite automata with their own schedule blocks.

- `BankCompositeSchedule01` test case. This test case verifies the simulation of a composite automaton with its own schedule block. This test case corresponds to the automaton *EnvBank* discussed in Section 3.1.2. The SIMUATOMATON parameter is set to *BankComposed*. This test case checks all of the necessary cases for simulations of composite automata with their own NDR block.

6.3 Parameterized Components

Now that we have tested the simulator's ability to handle composite automata with non-parameterized components we move on to expanding the test suite to check the more complex case of composite automata with parameterized components. Simulations of composite automata with parameterized components that use the component schedule block reuse strategy to resolve nondeterminism, involve the copying and tracking of multiple copies of the same schedule block. To validate the simulator's behavior in such cases we once again partition this section based on nondeterminism resolution strategy. Finally, since in the case of parameterized components, a composite schedule block may have a **with** block, we separate the composite schedule block nondeterminism resolution strategy into two cases, one that does not contain a **with**, and one that does.

6.3.1 Testing Reuse of NDR Blocks for Parameterized Components

The following tests check the behavior of the simulation of a composite automaton with parameterized components and no schedule block. They are based on automaton *Sys* from Section 3.2.1. They test the simulator's ability to handle components based on the same automaton and to maintain their independent states as they diverge during a simulation.

- `ProcChan01` test case. This is the most simple of the test cases. It involves only one component based on the *Channel* automaton and one component based on the*Process* automaton. This test verifies the parsing and simulation of a composite automaton whose parameterized components possess their own NDR blocks. The test ensures the simulator's ability to make independent copies of components based on their base automaton. It contains component invariants as well as composite ones. Further, it verifies the simulator's ability to track these copies as their state diverges. The components contain transition and action **where** clauses that are always satisfied. The correct interaction of the two components depends on the simulator's correct handling of constant parameters.

- `ProcChan04` test case. This test case builds on the previous one by adding two more components based on each of the base automata. This test validates the simulator's ability to copy components and their respective NDR blocks. In addition it tests component schedule blocks with diverging states and effect clauses with no program.

- `IGProcRelialbleChannel02` test case. The composite automaton in this case represents a reliable FIFO communication. The composite automaton here has two components based on one automaton and six based on another. The test case extends the above two by checking the ability of the simulator to handle cases where one component communicates with more than one other component based on the same base automaton.

6.3.2 Testing Composite Schedule Blocks for Parameterized Components

The following test checks the behavior of the simulation of a composite automaton with parameterized components, a schedule block, but no **with** block. It is based on automaton *Sys* from Section 3.2.2. In addition to testing the criteria of the above section, this test also checks the simulator's ability to handle component variable access and **fire** invocations in the composite schedule block.

- `ProcChan02` test case. The composite automaton in this case contains two components based on each of the base automata. The schedule block of the composite automaton accesses variables and invokes transitions of both components. Further, the test verifies the simulator's handling of constant formals in action signatures, an input action triggering the correct output action based on constant parameters, and a **where** clause in a transition that is violated and thus halts the simulation. It contains component invariants as well as composite ones.

6.3.3 Testing With Blocks

The following tests check the behavior of the simulation of a composite automaton with parameterized components, a schedule block, and a **with** block. They are based on automaton *Sys* from Section 3.3.3. In addition to testing the criteria of the above sections, these tests also check the simulator's ability to handle components declared in the **with** block.

- `ProcChan03` test case. The composite automaton in this case contains five components based on the *Process* automaton and four components based on the *Channel* automaton. This test case verifies the following points:
  - An input action **where** clause that causes an action that otherwise would have been triggered by an output action of another component, not to be,
  - Constant parameters, and correct output actions being triggered due to the constant parameters,
  - A precondition that fails and leads to the halting of the simulation,
  - Component invariants, and

  - Composite invariants.

- `IGProcRelialbleChannel01` test case. This automaton represents a reliable FIFO communication. In addition to testing the above points, this test case also verifies, a **for** loop in an `effects` clause and variable access via handle names in the composite schedule block.

- `WithSemantics01` test case. This test case verifies that all of the semantic checks for a **with** block in a composite schedule block discover the appropriate errors when those errors are present. These semantic checks are listed in Section 4.2.

Chapter 7

Application to Workflow

It is possible to use formal modeling to represent a wide variety of applications. The benefits of doing this are a more structured design, capability for invariant and theorem checking, and the ability to debug the system at design time.

The simulation of a single automaton allows for invariant checking. The simulation of paired automata allows for simulation relation checking. The direct simulation of composite automata, in addition to enhancing the formal modeling aspect of the simulator, allows for comprehensive debugging of distributed systems and applications. These functions together provide a useful tool for design time debugging of complex systems. One such example is workflow applications.

7.1 Workflow Description

Workflow applications explicitly model processes, most often but not limited to business processes. A workflow system implements and automates a process by modeling the flow of its states. "A workflow is simply a set of tasks that co-operate to implement a business process" [OW98]. Workflows abstract the user from a particular state by establishing an API to that state. Because of this, systems with distributed sources of information are well modeled by workflow systems [Cic98]. The following example from [MSK$^+$95] illustrates a workflow model of a part of a health care system.

Figure 7.1.1 models the flow of treating a patient at a hospital. Figure 7.1.2 models the diagnosis sub flow. When the `Diagnosis` block in Figure 7.1.1 is reached, the Patient workflow waits for a response from the Diagnosis sub workflow before deciding which way to proceed. The possible return values of the `End` block of the Diagnosis process are "Inpatient" and "Outpatient". The

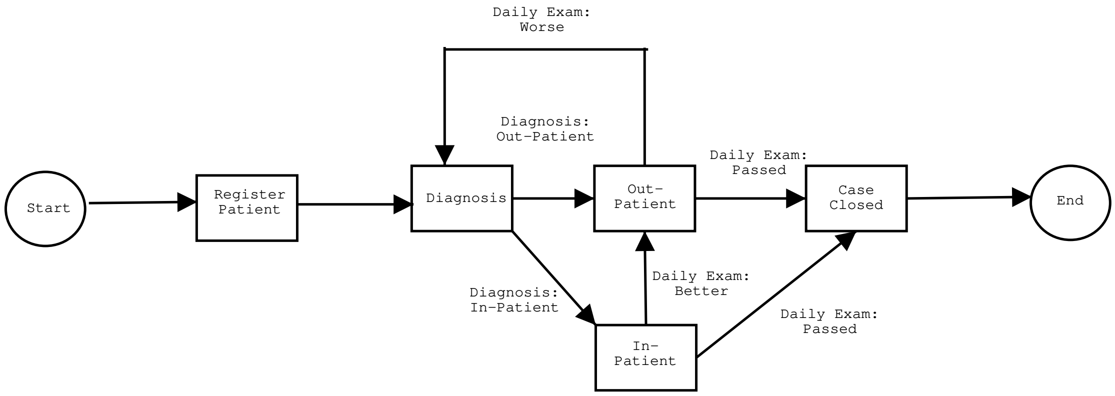

Figure 7.1.1: Workflow schema of a patient registration process

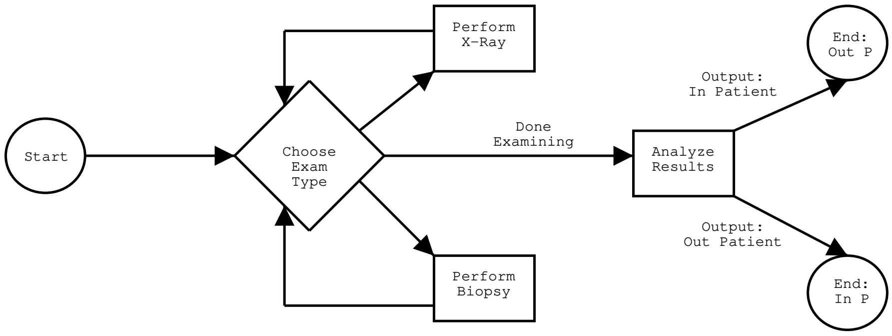

Figure 7.1.2: Workflow schema of a patient diagnosis process

Diagnosis schema leaves the mechanism for choosing what medical test (Biopsy, X-Ray) will be performed ambiguous. The schema simply declares that once the `Choose Exam Type` diamond is reached, the workflow will wait for input letting it know what examination was chosen. Later we will see how the automaton modeling the hospital provides this information.

While the Patient workflow is waiting for a response from the Diagnosis workflow, it may still communicate with other workflows of the systems. For example, another workflow may notify the Patient workflow of new information regarding the patient. The only cause of nondeterminism in this model is input from the system's environment, most likely from physicians. For a given set of input values from the physicians, the workflow is deterministic.

Figure 7.1.4 displays an automaton that models the Diagnosis workflow process from Figure 7.1.2. Figure 7.1.5 displays an automaton that models the medical activity at a hospital. This automaton provides an interface for the *Diagnosis* automaton to request that certain tests be performed on the patient being diagnosed. The *Hospital* automaton then communicates the results of the test back to the *Diagnosis* automaton. Other functions of the *Hospital* automaton are

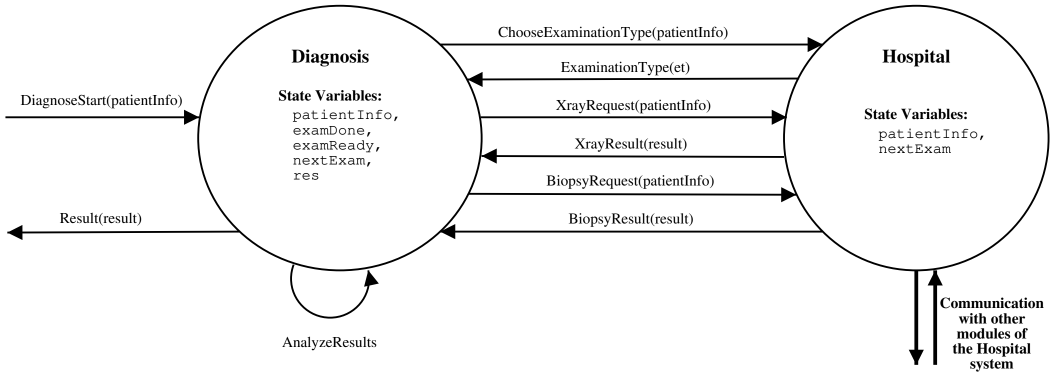

Figure 7.1.3: Interaction between *Hospital* and *Diagnosis* automata

not explicitly modeled here. They would include other medical processes such as surgery, physical therapy, blood transfusions, etc.

By composing the *Diagnosis* and *Hospital* automata, we can form a new composite automaton. Figure 7.1.3 shows the interaction between the two components of this automaton. We can now simulate this composite automaton and observe the full diagnosis cycle. Further, we can model other processes that use up the Hospital's resources as automata. By composing all of these process modeling automata, including the *Diagnosis* automaton, with the *Hospital* automaton, we can create a complex system automaton. By simulating this system automaton we can observe the work load placed on the Hospital by various modules of the system. We can thus see that automata simulation of worklfow processes is useful during the design time of the workflow processes. As it can reveal unintended and erroneous behavior in the workflow model as well as be used for resource allocation modeling.

7.2 Design Time Debugging of Workflow Systems

"Simulation can be used to study and refine workflow specifications. Because the workflow specification captures the implementation aspects of a business process model, their simulation and analysis can provide valuable feedback to the business process model evaluation [MSK+95]." Given an accurate model of the environment of the system, this feedback might consist of estimates of resource allocation. In the above example, a simulation might reveal the necessity for more hospital resources assigned to the support of the Diagnosis process. Further, a simulation of a complicated workflow schema can be beneficial in that it can expose a variety of properties of the workflow schema. [Cic98] These properties can be discovered through a simulation, possibly of a composite

```
Automaton Diagnosis
  type PatInfo = tuple of name: String, biopsy, xRay: Bool, sickLevel: Int;
  type ExamType = enumeration of X-Ray, Biopsy, Done, None
  type ResType = enumeration of InPatient, OutPatient
  signature
    input    DiagnoseStart(pf:PatientInfo), BiopsyResult(result:Bool),
             XRayResult(result:Bool), ExaminationType(et:ExamType)
    output   BiopsyRequest(pf:PatientInfo), XRayRequest(pf:PatientInfo),
             ChooseExamType(pf:PatientInfo), Result(result:ResType)
    internal  AnalyzeResults
    states   patientInfo : PatInfo :=null, examDone : Bool:=false,
             examReady : Bool:=false, nextExam : ExamType :=None,
             res : ResType :=null
  transitions
    input    DiagnoseStart(pf)
              eff patientInfo :=pf; examReady=true
    input    BiopsyResult(result)
              eff patientInfo.biopsy=result; examReady=true
    input    XrayResult(result)
              eff patientInfo.xRay = result; examReady = true
    input    ExaminationType(et)
              eff nextExam:=et
    ouput    BiopsyRequest(pf)
              pre nextExam=Biopsy ∧ examDone=false
    ouput    XrayRequest(pf)
              pre nextExam=Xray ∧ examDone=false
    output   ChooseExamType(pf)
              pre examReady = true      eff examReady = false
    output   Result(result)
              pre examDone= true ∧ result=res
    internal AnalyzeResultIn pre NextExam=Done
      eff if(patienfInfo.xRay=true ∧ patientInfo.biopsy=true)  then
             res=InPatient else res=OutPatient fi;
```

Figure 7.1.4: IOA specification of *Diagnosis* automaton

automaton, along with invariant checking. These properties are similar to some of those described in the I/O automata chapter of [Lyn96].

- *reachability* - can certain states be reached,
- *safety* - the workflow/state-machine does not terminate in an unacceptable state,
- *deadlock* - is it possible to reach a state where none of the prerequisites for its actions will ever be satisfied,
- *bottlenecks* - do certain states take up significantly larger amounts of times than the rest. The verification of this property would require the addition of the notion of time to the simulator.

```
Automaton Hospital
  type PatInfo = tuple of name: String, biopsy, xRay: Bool, sickLevel: Int;
  type ExamType = enumeration of X-Ray, Biopsy, Done, None
  signature
    input   BiopsyRequest(pf:PatientInfo), XRayRequest(pf:PatientInfo),
            ChooseExamType(pf:PatientInfo)
    output  BiopsyResult(result:Bool), XRayResult(result:Bool),
            ExaminationType(et:ExamType)
    states  patientInfo : PatInfo :=null, nextExam : ExamType :=None
  transitions
    output BiopsyResult(result) pre nextExam=Biopsy
            eff chosen :=choose x: Int where x>0 and x<11
              if(chosen<4) then result=true else result=false
    output XrayResult(result)   pre nextExam=X-Ray
            eff chosen :=choose x: Int where x>0 and x<11
              if(chosen<6) then result=true else result=false fi
    output ExaminationType(et)  pre nextExam=Done
            eff if(PatInfo.sickLevel>5) then et=Done
              elseif(PatInfo.sickLevel>8) then et=X-Ray
               elseif(PatInfo.sickLevel>11) then et=Biopsy fi;
    input BiopsyRequest(pf)    eff patientInfo=pf; nextExam=Biopsy
    input XrayRequest(pf)      eff patientInfo=X-Ray;  nextExam=Biopsy
    input ChooseExamType(pf)   eff patientInfo=pf; nextExam=Done
```

Figure 7.1.5: IOA specification of *Hospital* automaton

The majority of workflow systems are implemented through the use of databases, XML schemas, and PLSQL procedures. This tech stack does not lend itself as well to extensive simulation and debugging, as does the implementation of workflow via automata. There are two possible ways to improve the simulation and debugging capabilities of current workflow systems. One, we can enable a current workflow package to translate its representation of a schema to an IOA specification for the purpose of debugging and invariant checking. Two, implement the entire workflow software with IOA automata.

The benefit of the first option is that it does not require any modifications to be made to the particular implementation of the workflow system. Its only difficulty is the design of the tool that will convert the workflow schema to an IOA specification. Option two avoids the need for such a tool, but it is problematic for two reasons. Some workflow instances might have long life cycles and will require a mode of permanent storage such as a database. Also, option two requires a complete overhaul of the current workflow implementation.

I believe that option one is superior to option two because the schema to IOA translator tool should not be particularly difficult to implement. Further, the particular workflow implementation itself may be left unchanged. Also, in the absence of the schema to IOA translator tool, it is plausible that workflow designers may first model their systems with IOA, debug them at design time using the simulator, and then create the actual schema based on the modified IOA models that resulted after thorough debugging/simulating.

Chapter 8

Relation to Existing Features and Future Work

In this chapter we first discuss how the extension of the simulator to support composite automata has effected other features of the simulator. We then give suggestions for future work. The suggestions apply to both primitive and composite simulations.

8.1 Invariant Checking

In Section 1.1.1 we mentioned that the simulator supports an invariant checking feature. The user may enter a boolean predicate on the state variables of the automaton being simulated. This predicate is tested after every simulation step, and an appropriate error message is displayed whenever the predicate evaluates to false. During a composite simulation, the simulator checks the invariants of all components of the composite automaton. The simulator also has the capability of checking invariants written specifically for the composite automaton. The syntax for referencing state variables in an invariant of a composite automaton is the following:

```
CompositeAutomatonName.ComponentName.StateVariableName
```

Further, the predicates of invariants of primitive automata, may only refer to variables of that one automaton. In the case of invariants of composite automata, the predicate may refer to any of the state variables of any of the components. Thus an invariant relating the state variables of several components is legal. For example, consider the following two invariants of automaton *Sys* described in Section 3.2.2:

```
invariant nonEmpty of Sys:
  size(Sys.C1.contents)=0
invariant C1greaterC2 of Sys:
  size(Sys.C1.contents)>size(Sys.C2.contents)
```

Invariant `nonEmpty` is a predicate on the state variables of component *C1* only. Invariant `C1greaterC2` relates the state variables of component *C1* to component *C2*. References to component variables in invariant predicates of composite automata that contain a **with** block, parallel the references to these variables in the schedule block of such an automaton - the handle name is used as the component name. For example, the following is a possible invariant of automaton *Sys* described in Section 3.3.3 (`Proc1=P[1]` and `Proc2=P[2]` are declarations in the **with** block of that automaton):

```
invariant toSendEquals  of Sys:
 Sys.Proc1.toSend=Sys.Proc2.toSend
```

8.2 Paired Simulation of Composite Automata

The extension of the paired simulator to support composite automata has not been implemented, and is a future work. Consider the case where a complex specification is implemented via an algorithm described as a composite automaton. We now want to verify the correctness of the algorithm by writing a simulation relation from it to the specification. To evaluate this relation using the paired simulator, we need to be able to input a composite automaton as the implementation automaton. The following changes need to be made to allow such input to the paired simulator.

Currently the semantic checks on the IOA parser side require both the specification and the implementation automata to be primitive automata. These checks need to be relaxed. The rest of the code that runs the semantic checks casts the specification and implementation automata as primitive automata Java objects. This casting needs to be changed to allow composite automata. On the simulator side, the paired shell and the paired simulator need to be updated to account for the possibility of composite automata.

8.3 NDR Relinquish Control Command

In Section 2.1.3 we discussed the possibility of a component schedule block causing an infinite loop during a simulation of a composite automaton that reuses component schedule blocks to resolve nondeterminism. We proposed two solutions to this problem and described the implemented one, a limit on the maximum number of steps that a schedule block may run without returning an action to fire. Another alternative to solving the looping pitfall is the introduction of a new keyword to the syntax of schedule blocks. When a schedule block would encounter this keyword, it would relinquish control back to the high-level scheduling policy in a composite simulation, and halt the simulation in the primitive case.

8.4 User Interactive Nondeterminism Resolution

As an alternative method to NDR programs, this extension would allow the user to resolve nondeterminism as it occurs during the simulation. Prior to beginning the simulation the user will have to specify the choice of this option as opposed to the use of an NDR program. Then at runtime whenever the simulation is halted by either implicit or explicit nondeterminism, the user will be prompted to choose which branch the simulation should take. The user will also have the ability to inform the simulator that similar decisions should be made in the future without prompting the user again.

In the case of implicit nondeterminism, the user will be presented with a list of actions with satisfied preconditions, if possible. The user will then choose one of these actions. To aid the decision process, it will be useful to present the user with certain heuristics associated with each valid choice. The heuristic may be a partial snapshot of the resulting global state given that the particular action is taken. It could also be an estimate, either time or step amount, of simulation duration remaining until a certain state is reached, given that the particular branch is chosen.

In the case of explicit nondeterminism, the user will be presented with the **choose** statement that is responsible for halting the simulation. The user will then enter a value in the appropriate range for the variable in the **choose** statement. Once again, some heuristic of the possible choices should be presented to the user. In the case where this is not practical (displaying heuristics for every integer between 1 and 100 is not practical for `eff a := choose x: Int where x>0 and x<101`), heuristics can be displayed for a subset of all valid choices.

8.5 Debugging Tool

The ability of the simulator to print out trace logs and state snapshots is naturally extended to the creation of a more comprehensive debugging tool. This tool would comprise of features that are both common to most debugging applications and those that are more IOA specific. They will include runtime user interactive nondeterminism resolution, described in Section 8.4, step-through execution and breakpoints, and interactive execution logs.

Once this debugger tool is implemented the user will have a spectrum of tools that will enable him/her to explore in detail many possible executions of the system being modeled. The user will also be able to conveniently modify the system until he/she is satisfied with the observed simulations. This will allow the user to foresee problems in the system at design time and will aid the user in augmenting and improving the system at design time.

8.5.1 Step Through Execution and Breakpoints

This feature would provide the user with more control over the flow of the simulation. Both notions, stepping through a simulation and setting up breakpoints, are common to every standard debugging tool. Instead of only being able to observe the complete trace log after an execution has completed, the step through feature will enable the user to observe the state of the simulation as the trace log is being created. More specifically, after each addition to the trace log, the simulation will pause and present the user with the trace log up to this point and the global state of the system. The simulation will only resume once the user signals that he/she has completed analyzing the current situation and is ready for the simulation to continue.

Breakpoints will allow the user to choose, prior to the simulation, either one or many actions of the automaton being simulated. During the simulation whenever one of these actions is about to be fired, the simulation will pause and present the user with the trace log up to this point and the global state of the system. Once again, the simulation will only continue once the user signals that he/she is ready. Thus breakpoints will allow for a step through simulation that pauses at specified points of the simulation as opposed to pausing at every step of the simulation, as is the case with a standard step through execution.

To enhance step through simulation, the user will have the ability to select what variables are displayed as part of the state. Thus allowing him/her to concentrate on specific variables instead of having to look through the entire global state of the system.

8.5.2 Interactive Execution Logs

Once the features of step through execution, Section 8.5.1, and user interactive nondeterminism resolution, Section 8.4 are implemented, it would be useful to enhance trace logs to allow for simulation navigation. The user will be able to choose a point in a given trace log and return the simulation to this particular point. The simulation will then continue from this point.

This feature is particularly useful when coupled with user interactive nondeterminism resolution. It allows the user to return to a point in the simulation prior to some nondeterminism resolution decision. He/she can then make a different decision and follow the simulation to see how the branches differ. The user can repeat this until he/she has explored all of the desired branches of nondeterminism.

8.6 Graphical Improvements

The extension of the simulator's user interface from a text based one to a more graphical one would be a useful improvement. Combined with the features described above, a graphical interface would provide an engaging debugging environment. Via coloring, it would allow for comprehensive representations of the components of a composite automaton. Suggestions for the implementation of a graphical user interface to the simulator are mentioned in [Che98].

Appendix A

IOA Parser File Modifications

This chapter describes the changes made to the code of the IOA parser. The sections are partitioned by directory and a distinction is made between the files that were modified and the files that have been created.

A.1 Modified Files - parser

- **componentNode** - added two methods to modify this object after creation as is necessary for creating components declared in the **with** block. Methods : `setFormals` and `changeName`. - 1.7

- **compositionNode** - added `getAutNameByCompName` and `getComponentByName` methods and hashtables to support them. The former retrieves the automaton name of the component with the specified component tag. The latter retrieves the `componentNode` object for the component with the specified component tag. Both are used in `checkComposition`. Added schedule object to the *makeAbstract* method. Overloaded *set* method to allow for a schedule parameter. Added a `detScheduleNode` field to store the schedule for this composite automaton. Passes the `scheduleNode` to the composition upon *makeAbstract*, depends on `automaton/composition`(1.26) for the latest *make* method. - 1.9

- **detFireNode** - the fire statement might now have a component name associated with it. Depends on `automaton/ndrfire`(1.13). - 1.13

- **detScheduleNode** - added an `withNode` field to store a with blcok that may be associated with this schedule node. Overloaded *set* methods to allow for the passing of the with block. Added the with block to the *makeAbstract* method. Depends on `withNode`(1.1), `automaton/schedule`(1.9), `automaton/declaration`(1.1). - 1.15

- **grammar.src** - updated production rules for compositions to allow schedule blocks. Composite schdule blocks differ from primitive ones in that their invocation calls must have a component name prefix. Introduced production rules for the new notion of with blocks and declarations for composite schedule blocks. New WITHCOMP keyword. Depends on latest version of `compositionNode`(1.8), `detFireNode`(1.13), `detScheduleNode`(1.15),

  `compDetFireNode`(1.1), `withNode`(1.1), and `declarationNode`(1.1). - 1.23

- **invocationNode** - added `ltoken` field to store component name that may be associated with this invocation. Added accessor for this field and made sure it copies over when thi sobject is copied. - 1.2
- **ioaTokenizer** - adde the WITHCOMP = "with" keyword. - 1.10
- **Makefile** - added three new files : `withNode.pj`, `declarationNode.pj`, and `compDetFireNode.pj`. - 1.26

A.2 New Files - parser

- **compDetFireNode** - extends `detFireNode` and provides analogous *set* methods that allow for the passing of a component name associated with this fire statement. Depends on `invocationNode`(1.2). - 1.1
- **declarationNode** - node representing a single declaration line in a composite schedule block with block. Fields include: the component tag name, the handle name, and the actuals for this component. Depends on `automaton/declaration`(1.1). - 1.1
- **withNode** - node representing the entire with block of a composite schedule block. Consists of an array of `declarationNode` objects . *getCompName* method retrieves the component tag of the component with the specified handle name. Method used in `checkComposition`. Depends on `declarationNode`(1.1), `automaton/declaration`(1.1). - 1.1

A.3 Modified Files - automaton

- **component** - updated *toSValue* method to include formals for the component tag and the name of the base automaton that this component is based on. - 1.19
- **composition** updated the *make* method to take in a `schedule` parameter. Added the converison of the schedule to an `SValue` in the *thisKindOfAutomatonToSValue* method. - 1.26
- **formal** Added the keyword CONST (ILParser.KEYW_CONST) at the head of the `SList` of the IL representation of an action formal. Previously, there was no way to distinguish a constant parameter from a non constant one. - 1.8
- **Makefile** added new `declaration.java` file. - 1.19
- **ndrfire** added field to store component name in the case of this fire statement occuring in a composite schedule block. Added the component name to the *toSValue* method. - 1.13
- **schedule** added declarations field to store the declarations in a with block that may be associated with the schedule in the composite case. Overloaded constructors to include this field. Added this field to the *toSValue* method. Depends on `declaration`(1.1). - 1.9

A.4 New Files - automaton

- **declaration** represents an abstract declaration block in a with block of a schedule block of a composite automaton. FIX: *print* methods. - 1.1

A.5 Modified files - checker

- Moved *check* methods from `checkBasicAutomaton` to `checkAutomaton`, changed the accessibility of these methods to **protected** instead of **private**. This allows `checkComposition` to inherent the *check* methods it now needs to check the composite schedule block.

- **checkComposition** - *checkDetFire* overloaded to account for component name prefix in **fire** invocations of composite schedule blocks. Added *checkWithBlock* method that performs semantic checks on the **with** block; builds the *compNodes* array, which is later used by *extractStates* to make variables out of the components declared in the **with** block; extends the symbol map to make the sort of the components declared in the **with** block to be aggregate over the state variables of the automaton that its based on. The *extractStates* method has been extended to make the components declared in the **with** block in addition to those declared in the **components** section, into variables. The *checkCompNDRStates* calls the superclass *checkNDRStates* method and also makes sure that state variable names do not clash with the handle names established in the **with** block. Depends on `checkAutomaton`, all of the parser modifications. - 1.22

Appendix B

Simulator File Modifications

This chapter describes the changes made to the code of the simulator. The sections are partitioned by directory and a distinction is made between the files that were modified and the files that have been created.

B.1 Modified Files

- **ActualAutomaton** *abstract class* - now implements `ActualAutInterface` and supports all of its methods. - 1.12

- **Actual Transition** - Now knows whether it is a connecected transition or initial one. Contains string buffer for connected output. `AnnouncerExec` only registers initial transitions which in turn output all of the ones connected to them. - 1.19

- **DetActualAutomaton** - New *isSimulatable* variable is used in composite simulations to let the composite automaton know that this automaton will never again have any enabled transitions. - 1.4

- **ExecControl** allow ExecControls to be made for automata other than the one actually making it. (During a composite simulation, the composite automaton needs to create an NDR schedule for one of its components). Overloaded *execute* method with one that takes an `int` parameter. If the execute loop is not broken after this number of steps, an exception is thrown. - 1.13

- **FireProduct** - implement the new interface. Code to support composite schedule blocks. - 1.2

- **NDRActualAutomaton** creation of schedule control object needs to know which automaton it is to simulate. *nextTransition* method now calls the execution of the NDR program with a max-steps parameter. If the NDR program does not return the next transition before these steps have been exhausted, the NDR program stops looking for a transition. - 1.4

- **SimAutomaton** *Interface* - added method to create a new `ActualAutomaton` from this basic one. - 1.2

- **SimPrimitiveAutomaton** - now returns an `ActualAutInterface` instead of

`ActualAutomaton`, allowing the abstraction of primitive versus composite atuomaton for simulation. Is now capable of creating an independent copy of itself used when multiple components of a composition are based on the same automaton. Depends on interface `Copyable`(1.1) and the classes that implement it. - 1.14

- **SimILFactory** - added method for creation of `newCompositeAutomaton`. Overloaded *newState* method to allow a variable to know which automaton it belongs to. This allows the automaton name to be displayed in front of its variables. - 1.17
- **SimNDRFire** - modifications to allow composite schedule blocks. - 1.6
- **SimState** - Now knows its owner automaton's name, for display purposes. - 1.2
- **Simulator** - depends on `ActualCompositeAutomaton.java`(1.1) and updates to account for the new `ActualAutInterface`(1.1) interface. Also depends on `SimPrimitiveAutomaton`(1.12). - 1.31
    1. Interface Updates
    2. *doStep* modified to handle connected actions as well as initially fired ones
    3. Initial actions are aware of what needs to be outputed for their connected ones
    4. Overloaded *newControl* when a composite might need a control thats not for itself
    5. Added static MAX_NDR_STATES variable which is defaulted to 500 and can be changed via a command line argument. This variable determines the maximum number of steps a particular call to an NDR program is executed.
- **shell/SimShell** - Interface changes. Lets `StepListener` know that the simulation is a composite one. Introduced new command line parameter, **ndrSteps**. This parameter specifies the maximum number of steps to run a particular call to an NDR program. This variable is statically stored in `ioa.simulator.Simulator` and is defaulted to 500. - 1.64
- **shell/StepListener** - Output automaton name next to state variable when the simulation is a composite one. Output all of the connected actions when handling an initial action that has actions conncected to it. - 1.19

B.2 New Files - Simulator

- **ActualAutInterface** *Interface* - new interface on the actual side to abstract away knowledge of composite vs primitive automaton. Can ask it whether the automaton is a composite one or not. - 1.1
- **ActualCompositeAutomaton** - 1.2
    1. Dependency - `SimCompositeAutmaton`.
    2. *NextTransition* method looks through components on search of next action to fire. Two possible non determinism resolution strategies. One - use the blocks of the components. Two - use an NDR block specifically defined for the composite automaton.
    3. *fireConnected* method looks for input transitions that might possibly be connected to the recently fired output transition. Executes all such transitions.

4. Lets connected actions know that they are connected. Adds the output produced by connected actions to the action that initially triggered them.
5. *componentSelectionPolicy* variable determines the order in which component automata are tested for enabled transitions. Random and Uniform policies have been implemented. A Weighted policy is being implemented. Currently, the uniform policy is the default option.
6. To do: implement *toSValue*.

- **Copyable** *Interface* - objects that implement this interface are capable of copying their reprsentations. This is used in compositions where multiple components are based on the same automaton. Classes that implement this interface:
  - SimApplicationTerm 1.13
  - SimExistsTerm 1.4
  - SimForAllTerm 1.5
  - SimLiteralTerm 1.4
  - SimVarRefTerm 1.7
  - SimAssignment 1.7
  - SimChoice 1.11
  - SimConditional 1.4
  - SimPairedFire 1.7
  - SimNDRFire 1.7
  - SimNDRWhile 1.6
  - SimNDRYield 1.4
  - SimNOP 1.5
- **SimCompositeAutomaton** parallel of *SimPrimitiveAutomaton*. - 1.1
- **StepsExceededProduct** thrown when a particular NDR program has been running for more steps than the alloted number. Used for control relinquishing between component NDR programs in composite simlations. - 1.1

B.3 Modified Files - il

- **AutComponent** - made fields and accessor methods. Now extends *BasicILElement*. Added fields to store the actuals of a component declared in a **with** block and the formals of the component tag corresponding to this component. Its constructor has been overloaded to allow the passing of these fields. To do: fix *toSValue* method. - 1.3
- **BasicAction** formals are now represented as *Term* objects as they may be constants. - 1.9
- **BasicActionTable** Added method to allow `BasicCompositeAutomaton` to return action table. - 1.11

- **BasicCompositeAutomaton** Changes dealing with new `AutComponent` methods. *getActionTable* update. - 1.6
- **BasicILFactory** Overloaded *newState* method to allow state to know which automaton it belongs to, for display purposes. - 1.9
- **BasicState** Allow state to know which automaton it belongs to, for display purposes. Related to *SimState*. - 1.7
- **BasicVariable** To allow *StepListener* to display information about a connected action, modified a method. - 1.7
- **CompositeAutomaton** synchronized *addComponent* method with the latest version of *AutComponent*. - 1.4
- **ILFactory** *abstract class* Added *newState* method to allow creation of a variable that knows which automaton it belongs to. This enables the output of the automaotn name in front of the variable name. - 1.10
- **ILParser** - 1.43
  1. added *curAut* variable to keep track of the automaton currently being parsed,
  2. *parseState* and *parseStates* modification to allow a state (variable) to know what automaton it belongs to,
  3. *parseTerm* is aware of the possbility of a constant term,
  4. *parseCompositeAut* begins the handling of parsing a composite schedule block.
  5. The `TempComponent` internal class is ised to store the temporary representation of a component when it is described in the **components** section. It is later used when `NDRCompositeAutomaton` parses the with block of this composite automaton to instantiate components based on this temporary representation.
- **HookILFactory** Added *newState method*; implementing that of the *ILFactory* interface. - 1.7
- **NDRFire** Modification to allow composite schedule block and storing of the component name prefix in composite schedule blocks. - 1.6
- **NDRILFactory** handles the parsing of a **fire** statement that may appear with a component name prefix in a comosite schedule block. - 1.5
- **PrimitiveAutomaton** and **BasicPrimitiveAutomaton** added copy method that is used by composite automata that have more than component based on the same base automaton. The meaningful implementation of this method is in

  `simulator/SimPrimitiveAutomaton` and overrides the implementation in

  `BasicPrimitiveAutomaton`. - 1.5
- **Variable** *interface* Added *setAutName* and *getAutName* methods to allow variables to know what automaton they belong to. - 1.4
- Files modified to allow the marking of terms as constant formals of action signatures:

- **BasicApplicationTerm** - 1.10
- **BasicExistsTerm** - 1.5
- **BasicForAllTerm** - 1.7
- **BasicLiteralTerm** - 1.5
- **BasicSortRefTerm** - 1.5
- **BasicVarRefTerm** - 1.5
- **Term** - 1.5

B.4 New Files - il

- **NDRCompositeAutomaton** parallel of `NDRPrimitiveAutomaton`. Handles the parsing of the composite schedule block and the with block that may appear there. - 1.2

B.5 Files where the only changes involve the naming of the new interfaces

- **codegen/ig/InvocationGenerator** - 1.3
- **codegen/ig/InvocationListener** - 1.2
- **il/ILUnparser** - 1.31
- **simulator/daikon/DaikonListener** - 1.18
- **simulator/daikon/DeclsPrinter** - 1.18
- **simulator/daikon/PairedDaikonListener** - 1.6
- **simulator/daikon/SplitterWriter** - 1.4
- **simulator/PairedFireProduct** - 1.2
- **simulator/PairedImplAutomaton** - 1.12
- **simulator/PairedSimulator** - 1.6
- **simulator/shell/PairedShell** - 1.32
- **simulator/shell/PairedSteplistener** - 1.13

B.6 Test Suite

- **Test/Makefile.common** Added SIMAUTOMATON parameter for sim testing. When specified in a `Makefile` of a test case, this parameter determines which automaton is to be simulated. This is necessary for testing of composite simulations as the ioa file may contain more than one automaton. - 1.29

- **Test/Makefile** Support for the SIMAUTOMATON parameter for sim testing. When specified in a `Makefile` of a test case, this parameter determines which automaton is to be simulated. This is necessary for testing of composite simulations as the ioa file may contain more than one automaton. - 1.29

Appendix C

IOA Grammar

C.1 Description

The .ioa file that is the input to the checker gets parsed according to the IOA grammar. This grammar is defined in */Code/ioa/parser/grammar.src*. The grammar consists of two parts, tokens and rules. The tokens themselves are divided into two parts, terminal tokens and non terminal tokens. The terminal tokens are the leaves of the parse tree, while the non terminal tokens are the non-leaf nodes of the parse tree. Each non terminal token must appear on the left side of a rule.

The grammar gets processed by the *javaCup* tool. This tool creates files, described in detail below, that act as the parser for the IOA language.

C.1.1 Tokens

Terminal tokens are associated with keywords of the IOA language; notions that do not need to be further broken down. A terminal token is represented by an `ioa.parser.ltoken` object. It is declared using the following syntax,

```
terminal ltoken tokenName
```

For example, punctuation marks such as a comma and a semicolon, key words such as **automaton** and **input**, and operators such as **or** and **and** are all terminal tokens.

Non terminal tokens are associated with notions that need to be further broken down. A non terminal token is represented by a specific subclass of the `ioa.parser.Node` object. It is declared using the following syntax,

```
non terminal className tokenName
```

For example, the high level notions of an IOA spec, an automaton definition, and a transition definition are all non terminal tokens. Figure C.1.1 displays the specification of automaton *Fibonacci*. After parsing, everything other than `automaton Fibonacci` would be represented by the non terminal token *basicAutomaton*. This token would further be broken down according to the rules described in Section C.1.2.

```
automaton Fibonacci
  signature
    internal compute
  states
    a:Int := 0,
    b:Int := 1,
    c:Int := 1,
    d:Bool
  transitions
    internal compute
      eff a := b ;
          b := c ;
          c := a + b; }
```

Figure C.1.1: *Fibonacci* automaton

C.1.2 Rules

Rules of the IOA grammar are of the form:

non-terminal-token ::= +non-terminal-token/terminal-token method-call()
(where + denotes, one or more)

Rules define the connection between non terminal tokens and other non terminal or terminal tokens. The highest node in the parse tree is represented by a start token. This non terminal token is the only one that does not appear on the right side of any rules. In our case this happens to be the *spec* token. The rule of the IOA grammar that has *spec* on its left hand side, defines the notions that may make up a spec object. Other rules further refine these notions until everything is represented by a terminal token. For example, the following rule specifies the notion of a basicAutomaton:

```
basicAutomaton ::=

SIGNATURE:l formalActions:a states:s transitions:t
            {set(l,a,s,t)} |
SIGNATURE:l formalActions:a states:s transitions:t tasks:tk
            {set(l,a,s,t,tk)} |
SIGNATURE:l formalActions:a states:s transitions:t schedule:sc
            {set(l,a,s,t,sc)} |
SIGNATURE:l formalActions:a states:s transitions:t tasks:tk schedule:s
            {set(l,a,s,t,tk,sc)} ;
```

The non-terminal token *basicAutomaton* represents the main body of a primitive basic automaton. The above rule defines four possible ways that the basicAutomaton may further be broken up. The terminal token SIGNATURE, represents the IOA keyword **signature** and is the required beginning in each of the four possibilities. The other tokens are all non terminals. The first case is an automaton specification without tasks and a schedule block, the second with tasks but no schedule block, the third with a schedule block and no tasks, and finally the fourth is an automaton with tasks and a schedule block.

Each token is followed by a ":" and a temporary variable assignment. These temporary variables are used in the method calls that follow each one of the cases. In the declaration section of the non terminals, the following line can be found:

non terminal basicAutomatonNode basicAutomaton;

This declaration of the *basicAutomaton* token indicates that the basic automaton notion is represented by a `basicAutomatonNode` object. The methods that are specified at the end of each of the four cases are methods of this object. When this rule is triggered, a `basicAutomatonNode` is created and depending on which case was matched, the corresponding method is called. The arguments to these methods are the temporary variables whose type depends on the token that they represent. Thus in the first case, `set(l,a,s,t)`, l (SIGNATURE) is an `ltoken`, a (formalActions) is a `factoredListNode` of `actionNode` and `Node` (described in Section C.1.3), s (states) is a statesNode, and t (transitions) is a listNode of transitionNode's.

C.1.3 Typed Lists

Some of the tokens are declared to be of type ListNodeXX*name*YY or FactoredListNodeXX$name_1$ZZ$name_2$YY. The former denotes a typed list of *name* objects, while the later denotes a typed list of $name_1$ objects whose members are in turn lists of $name_2$ objects. The typed lists are only supported by polyj and are not recognized by javaCup. Thus they are declared with the XX, YY, and ZZ delimiters. These are later converted to polyj representations by the postprocessor (described below). For example, the following declarations of the non terminal token **operators**, denotes that the operators token is represented by a list of operatorNode objects.

non terminal ListNodeXXoperatorNodeYY operators;

C.1.4 Processing the Grammar

The Makefile for the */Code/ioa/parser* directory contains scripts that do all of the following. The `grammar.src` file gets preprocessed into `grammar.cup`. This file is processed by javaCup and two files are produced, `parser.java` and `sym.java`. These files now get post-processed into polyj files to allow for the use of parameterized lists, `lparser.pj` and `sym.pj` are produced. Finally, these two files get compiled into what becomes the parser tool. This process is described in more detail in the `grammar.src` file.

C.2 Auxiliary files

There are a few other files that are involved with the IOA parser other than `grammar.src`. These files define the IOA keywords and create a mapping between them and the representation created for them by javaCup.

notions/lexical.java this file defines the actual text of keyword strings. It is updated manually.

parser/sym.pj this file is generated by the javaCup process. It contains the internal representations of the tokens

parser/tokenizer.pj this file is manually updated. It is the link between the keywords in `lexical.java` and `sym.pj`.

parser/ioaTokenizer.pj same as above, ioa specific

parser/lparser.pj this file is generated by the javaCup process. It is the actual parser.

C.3 Brief Guide to Modifying Grammar

If the modification required the addition of new keywords to the IOA language or the update of existing keywords in the IOA language:

- Update `lexical.java` by adding/modifying keyword strings, and
- Create connection between the keyword and its representation in the generated parser by updating `ioaTokenizer.java` (in the case of new keywords).

If the modification requires the creation of a new intermediary token or the modification of the behavior of one:

- Create/update the appropriate subclass of `Node.java`, and
- Verify that the methods you intend to call upon the processing of this object during parsing have the intended signatures/behaviors.

In all cases:

- Update grammar.src to encorporate the new/modified keywords, tokens, and rules, and
- Recompile the the *Code/ioa/parser* and the *IOA_Toolkit/bin* directories to make all of these changes take effect.

Bibliography

[Che98] Anna E. Chefter. A simulator for the IOA language. Master's thesis, MIT, May 1998.

[Cic98] Andrzej Cichocki. *Workflow and Process Automation: Concepts and Technology.* Kluwer Academic Publishers, 1998.

[Dea01] Laura Dean. Improved simulation of input/output automata. Master's thesis, MIT, September 2001.

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

[KCD+02a] Dilsun Kirli Kaynar, Anna Chefter, Laura Dean, Stephen Garland, Nancy Lynch, Toh Ne Win, and Antonio Ramirez-Robredo. The IOA Simulator. Technical Report 843, MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA, July 2002. `http://theory.lcs.mit.edu/tds/ioa.html`.

[KCD+02b] Dilsun Kirli Kaynar, Anna Chefter, Laura Dean, Stephen Garland, Nancy Lynch, Toh Ne Win, and Antonio Ramirez-Robredo. Simulating nondeterministic systems at multiple levels of abstraction. August 2002. In *Tools Day* held in conjunction with CONCUR'02, Brno, Czech Republic.

[Kun93] Thomas Kunz. Distributed debugging – a case study. Technical Report TI–3/92, Institut fur Theoretische Informatik, Darmstadt, Germany, February 1993.

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

[MSK+95] John A. Miller, Amit P. Sheth, Krys Kochut, Xuzhong Wang, and Arun Murugan. Simulation modeling within workflow technology. In *Winter Simulation Conference*, pages 612–619, Arlington, VA, December 1995.

[OW98] P. D. O'Brien and W. E. Wiegand. Agent based process management: Applying intelligent agents to workflow. *The Knowledge Engineering Review*, 13(2), 1998.

[RR00] J. Antonio Ramirez-Robredo. Paired simulation of I/O automata. Master's thesis, MIT, September 2000.

[Tau03] Joshua Tauber. Definition and expansion of composite automata in IOA. PhD thesis, in progress. MIT, 2003.

[Yan92] Z. Yang. Global snapshots for distributed debugging: an overview. Technical Report TR-92-03, Laboratory for Distributed and Parallel Computing, University of Alberta, Edmonton, Alberta, Canada, March 1992.