Simulation of Timed Input/Output Automata

by

Panayiotis P. Mavrommatis

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

May 2006

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2006

Certified by..
Dilsun K. Kaynar
Post-doctoral Research Associate, Theory of Distributed Systems Group
Thesis Supervisor

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

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

Simulation of Timed Input/Output Automata

by

Panayiotis P. Mavrommatis

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This Master of Engineering Thesis describes the design, implementation, and usage of the TIOA Simulator. The TIOA Simulator, along with the other components of the TIOA Toolset aims to provide a framework for developing dependable distributed systems. The project is based on the Timed Input/Output Automaton framework, and supports TIOA, a formal language for specifying timed I/O automata. Simulation of TIOA programs is useful in the process of testing the proposed system over a specific set of executions. During the execution the Simulator is able to test proposed invariants and validate a proposed simulation relation between the system's implementation and its specification. A step correspondence between the steps of the implementation and the specification drives the validation of the simulation relation. The identification and validation of the invariants and the simulation relation constitutes the first step towards a formal verification of the system's correctness. The proposed step correspondence can be used in a formal proof to show that the proposed relation is indeed a simulation relation.

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

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

Acknowledgments

I would like to thank Prof. Nancy Lynch for giving me the opportunity to work with the Theory of Distributed Systems group and for the precious guidance throughout the past three years.

I was first introduced to the design and implementation of the IOA project by Joshua A. Tauber, with whom I worked at the final implementation stages of the IOA Code Generator. I then worked with Chryssis Georgiou at the University of Cyprus on using the IOA Toolkit to compile a number of distributed algorithms, an experience that introduced me to the process of specifying and developing complex distributed systems using a formal language such as IOA, as well as the issues involved in the simulation and compilation of such a language.

The TIOA group was then formed, headed by Prof. Nancy Lynch and Prof. Alexander Shvartsman, with the purpose of creating the TIOA Language and a set of supporting tools. The design and implementation of the TIOA Simulator was a result of my collaboration with Dilsun Kaynar and Fivos Constantinou and was based on the work of all the former IOA members that contributed to the IOA Simulator.

Steve Garland provided valuable help with the design of the simulator-specific language extensions, and provided the syntax and semantic checker. Hongping Lim contributed in the development and testing of the connection of the checker with the various back-end tools. Some of the first users of the TIOA Simulator that provided valuable feedback and suggested improvements to the tool were Wenkai Tan and Marios Assiotis.

Στην Αριάνα

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Designing and implementing dependable distributed systems is a challenging problem. With many mission-critical applications running in distributed environments, such as air traffic control or road traffic management, that have correctness and fault-tolerance requirements, and a range of distributed applications that may additionally require performance and availability guarantees, the problem becomes significant. On the other hand, as the complexity of such systems increases, providing these guarantees becomes harder.

To cope with these challenges, researchers have developed mathematical models that provide frameworks in which formal specification as well as proofs of correctness can be performed in a structured and effective manner. One of these frameworks is the Timed Input/Output Automaton framework [7], in which the basic building blocks are state machines with both discrete steps and continuous evolution of state. This framework has evolved from the Input/Output Automaton model [8], in which only discrete steps are allowed. The I/O Automaton model was used in a number of research papers and books in order to specify and verify several distributed algorithms without timing guarantees. The model has been supported by a formal language, IOA [5], and a number of software tools that aim to check the IOA specifications for syntactic correctness, simulate IOA executions [2, 9, 3, 15, 10], connect IOA to various theorem provers [1, 14], and compile IOA to Java [11, 13], thereby providing a method to implement a provably correct distributed system.

Recently, a formal language that supports the precise specification of Timed I/O Automata has been developed [6]. The language, called TIOA, is a natural extension of IOA. In an effort to provide a similar set of tools for TIOA as for IOA, some of the IOA tools have been extended to support TIOA. The Syntax and Static Semantics Checker tool, the PVS Theorem Prover Translator and the Simulator are the three main tools that were first extended for this purpose. When used together, these tools can be used to design and formally specify distributed systems in TIOA. The system can be specified with the help of the TIOA Checker, tested and analyzed with the TIOA Simulator, and be verified using the PVS theorem prover.

This Master's Thesis describes the design, implementation, and usage of the TIOA Simulator. The main purpose of the TIOA Simulator can be summarized as follows:

- Create and simulate specific "test-case" executions, which can provide feedback about the model's correctness and show potential pitfalls. Proposed invariants that can be used in proofs of correctness (both safety and liveness) can be shown to hold throughout these simulated executions.

- Test the interaction of the various system components by either creating specific "test-case" executions among the components or by running all the components independently. Once again, invariants can be shown to hold throughout such executions.

- Test proposed simulation relations, which form the basis for proofs of correctness, especially when the system tested is proposed to implement another, "higher-level" system. Frequently a high-level description of the system that models the requirements and guarantees of the system is first specified, and a lower-level implementation can then be verified by relating the two versions of the system using a simulation relation. Similarly, simulation relations can be used to relate an optimized version of an algorithm against a simpler, easier to verify but inefficient version of it.

1.2 Mathematical Framework

The essence of TIOA and its tools lies on the mathematical framework of the Timed Input/Output Automaton Model [7]. We provide a brief and informal description of the model here, and refer the reader to [7] for a complete and formal specification of the model. A Timed I/O Automaton is a state machine that can modify its state both instantaneously and continuously over time. Discrete *transitions* modify the automaton's state instantaneously, while *trajectories* specify how a set of the state variables evolves with time. We call the state variables that vary with time *time-dependent*.

Actions are either internal or external; internal actions are not visible to the automaton's environment, while external ones are. External actions can be either output or input ones; input actions model inputs to the system and are always enabled. Internal and output actions may have preconditions that specify under what conditions of the state they are enabled. The externally visible behavior of an automaton, its *trace*, is a sequence of alternating external actions and time-passage steps over the set of empty variables (the only information recorded in the these trajectories is the amount of time that passes).

1.2.1 Composition

A complex distributed system can be represented by composing automata that model individual parts of the system. Under certain compatibility conditions, the various components of the system can interact with shared actions. A typical client-server application, for example, can be modeled by a composition of the client automaton, the server automaton, and the channel automaton representing the network.

1.2.2 Abstraction

A common practice in reasoning about complex systems is to provide an abstract specification of the system along with the concrete implementation. Two separate automata can be used to specify both descriptions, and the concrete automaton can be shown to implement the abstract one using the notion of simulation relations.

Simulation Relation: Given a concrete automaton A and an abstract automaton B, one can show that A implements B if there exists a *forward simulation* from A to B, which is defined as a relation R between the states of A and B such that

- Every start state x of A is related to some start state y of B ($\forall x.\exists y.xRy$)
- For each step (transition or trajectory) `a` in A, starting from state x and ending in state x', there is a sequence of steps `b` in B, starting with y where xRy and ending in y' such that

$$[\text{trace(a)} = \text{trace(b)}] \wedge x'Ry'$$

1.2.3 Non-determinism

The Timed I/O Automaton model includes several sources of non-determinism: At any point in time, many transitions with many different possible parameter values may be enabled, and many trajectories might be possible to follow for many different durations. Moreover, explicit non-deterministic choices are allowed for parameters and statements.

1.3 The TIOA Language

The TIOA Language [6] is a formal language that can be used to specify Timed I/O Automata. We provide a brief introduction to the language by means of an example here, and refer the reader to the TIOA User Guide and Reference Manual [4] for a complete specification of the language.

Figure 1-1 shows an example automaton in TIOA. The `PeriodicSend` automaton uses the continuous variable `clock` as a timer to send a message every `u` time units. A `send(m)` transition can occur if `clock =u`, and when it does, the timer is reset (`clock :=0`). The trajectory definition `traj` specifies that `clock` evolves with the same rate as real time, and that the trajectory's execution must stop when it is time to send a new message. The types used in this example have their regular meaning. The type `AugmentedReal` is an extension of the type Real that also includes a value for positive infinity.

```
vocabulary Messages
  types M enumeration [nil, m1]

automaton PeriodicSend(u: Real)
  imports Messages
  signature
    output send(m: M)
  states
    clock: AugmentedReal := 0
  transitions
    output send(m)
      pre clock = u
      eff clock := 0
  trajectories
    trajdef traj
      stop when clock = u
      evolve d(clock) = 1
```

Figure 1-1: An example TIOA program

The basic module in the language is the **automaton**, which is specified by its **signature** (discrete step declarations), **states** (state variables), **transitions** (discrete step definitions) and **trajectories**. A number of built-in data types is provided and users can specify their own data types and operators in **vocabulary** constructs. Tuples, enumerations and unions can be specified, for example as in Figure 1-1. An automaton or another vocabulary can import these definitions using an **imports** statement. The automaton's states can be declared and optionally initialized in the **states** block, as in Figure 1-1.

Actions and transitions

The actions are declared in the **signature** block, where their type (input, output or internal), name and parameters (parameter name and type) are specified. In the **transitions** block the transitions are defined, with the optional preconditions (**pre**) and effect block (**eff**) for each transition. The precondition is simply a boolean predicate or a series of such separated by a semicolon (;), while the **eff** block expects a TIOA program, i.e. a series of statements separated by a semicolon. The following statements can be used:

- Assignments, of the form `s :=expression`, where `s` is a reference term to a state variable or a local variable.

- Conditionals, of the form **`if`** `pred` **`then`** `program`, optionally followed by a series of **`elseif`** `pred` **`then`** `program` and/or an **`else`** `program`.

- Loops, of the form **for** var:**type** **in** set **do** program **od** or

  for var:**type** **where** pred **do** program **od**.

Trajectories

The continuous transformation of state is specified in the **trajectories** block. There can be multiple trajectory definitions, and each one is of the following form:

```
trajdef traj_name
  invariant inv_predicate %optional
  stop when stop_predicate %optional
  evolve    ev_clause1; ev_clause2; ...
```

The name of the trajectory is given in `traj_name`, `inv_predicate` is a boolean predicate that must hold throughout the execution of the trajectory, `stop_predicate` is a boolean predicate indicating the condition that would cause the trajectory to stop: time cannot advance beyond the point at which the stopping condition becomes true. The set of `ev_clause`*i* includes differential and algebraic equations and inequalities describing the evolution of the time-dependent variables in the automaton.

Invariants

Inside a TIOA specification one can specify boolean predicates that must hold throughout the execution of an automaton. These can be specified in the form
invariant i_name **of** aut_name:
followed by a list of boolean predicates separated by a semicolon.

Composition

A composite automaton is specified by its components, which are given a name and a parametrization of their formal parameters. Figure 1-2 shows an example of a composite automaton, which consists of two `Process` automata and a `Channel` automaton.

Non-determinism

TIOA inherits the non-deterministic nature of the mathematical framework and includes two fundamental sources of non-determinism:

```
automaton Composition
  components C: Channel(1,2);
             P1: Process(1);
             P2: Process(2);
```

Figure 1-2: An example composite automaton.

1. The scheduling of transitions and trajectories: At any given point in the execution, any number of transitions might be enabled with many (possibly infinite) possible values for the transition parameters. Similarly, multiple trajectory definitions might be allowed to be followed, for possibly infinitely many different durations.

2. The explicit non-deterministic choices involving **choose** statements, **choose** parameters and **choose** expressions in initial assignments.

We refer the reader to [4] for a description of other elements of TIOA such as formal parameters, local variables, **ensuring** clauses, **where** clauses, function declarations (`let`), the **const** keyword, **initially** statements, **choose** statements, data type, and operator declarations.

1.4 TIOA Tools

The TIOA Toolkit contains the following tools that support TIOA:

- **Syntax and Semantic Checker:** The TIOA Checker can be used to check whether the specification follows the TIOA syntax and static semantics,

- **Simulator:** The TIOA Simulator, for simulating the system's execution and for checking invariants and simulation relations, and

- **Interface to the PVS Theorem Prover Tool:** For verifying the model's correctness and other properties.

We describe below how a distributed system can be specified, implemented, and verified using the TIOA tools. The system designer would:

1. Formally express the specification of the system. Using TIOA, the automata that model the system, the abstract state and any invariants that must hold on the the abstract state can be written. Safety properties can typically be expressed as

invariants, for example. The TIOA Checker can perform static syntax and semantics analysis on the specification. It may also be desirable to simulate some executions of the system at this level of abstraction. Simulation will provide a set of traces that can help understand the specification and make any improvements, corrections, or simplifications.

2. Provide an implementation for the system. Using TIOA, an actual implementation for the system can be written and tested, in a similar way to the specification. The implementation may involve invariants of its own. Testing the implementation with the TIOA Simulator may reveal bugs, problems, improvements, or increase the confidence about the correctness of the implementation.

3. Show that the implementation satisfies the specification. The framework provides the notion of simulation relations for this purpose. The relation between the states of the two automata can be specified in TIOA, along with a "step correspondence" to allow the Simulator to check this relation over the executions the user provides.

4. After testing the system and the simulation relation over some user-specified executions, an exhaustive testing may be desirable using model-checking techniques. A future tool will either extend the TIOA Simulator to implement model-checking techniques or translate the TIOA specifications to UPPAAL or another language that can be model-checked, either fully if the program is finite-state or probabilistically.

5. In many occasions a full formal proof is desired. The framework itself provides a number of techniques (such as simulation relations) that can be very helpful in this task. Moreover, the TIOA to PVS Translator can translate TIOA to the PVS Theorem Prover, a tool that facilitates mathematical proofs. The verification will end up proving the simulation relation and the invariants that were specified and tested in the simulation. The step correspondence that was specified in order to drive the paired simulation will constitute the basis of the proof.

6. After verifying the system's correctness to the degree desired, a possible future tool would execute the system, for example by first translating the TIOA implementation to a Java one. Such a tool could be an extension of the already existing IOA Compiler [11], which compiles IOA to Java.

1.5 Previous Work

The TIOA Simulator evolved from the design and implementation of the IOA Simulator, first designed and developed by Anna Chefter [2]. Antonio Ramirez's work [9] extended the Simulator to introduce the Non-Determinism Resolution language as well as paired simulations and thus the ability to test simulation relations among two automata. Laura Dean [3] enhanced the Simulator and introduced a connection to the Daikon invariant detection tool. Ed Solovey's M.Eng thesis [10] extended the tools to support simulation of composite I/O Automata. Michael Tsai's M.Eng thesis [13] introduced a common methodology for simulating and generating Java code for IOA, and a way to allow users to define their own data types.

1.6 Organization of this Thesis

Chapter 2 explains which restrictions and extensions to the TIOA language are necessary to allow simulations, and describes these restrictions and extensions in detail. In Chapter 3 we discuss how simulation of primitive and composite automata is achieved, and how we perform paired simulations between two related automata. We illustrate the restrictions and extensions to TIOA as well as the output of the Simulator with the use of an extended example (timeout-based failure detection) in Chapter 4. We discuss some of our design decisions, as well as alternatives and future work in Chapter 5, and conclude in Chapter 6.

Chapter 2

Design

Apart from simply executing a set of automata, the TIOA Simulator can test proposed invariants and validate a proposed simulation relation between a low-level "implementation" automaton and the high-level "specification" one. The identification and validation of the invariants and simulation relations constitutes the first and perhaps the most important step towards a formal verification of the system's correctness. The proposed step correspondence that will drive the paired simulation constitutes the basis of a formal proof. Equally important is the fact that the the simulator's output, the trace of the execution can be used to test some aspects of the system's performance such as message complexity.

To accomplish the goals mentioned above, the TIOA Simulator provides the following features:

- Execution of primitive automata,
- Execution of composite automata, and
- Paired execution of two related automata.

There are various issues related to the execution of TIOA, such as the implementation of data types, the initialization of the state variables for the various automaton instances, etc. These issues are not very hard to deal with, however. On the other hand, the non-deterministic nature of the framework makes simulation of TIOA programs without any modifications very hard. This is mainly because we are attempting to execute programs written in a language that is primarily for specifying the behavior, using logical predicates and non-determinism. Such features of the language, and in particular the following ones are the hardest problems to solve automatically:

- Differential equations in evolve predicates, arbitrary stopping conditions, arbitrary existential and universal quantifiers, and
- Non-determinism, either from the scheduling of transitions and trajectories or from explicit **choose** statements.

Our solution to the above problems is to both restrict the language to a subset of the language we can simulate and let the user resolve the non-determinism on a case-by-case basis. In particular, we

- Restrict TIOA to a language that can be simulated, and
- Extend TIOA with syntax that can be used to resolve non-determinism.

In Sections 2.1 and 2.2 we elaborate on the TIOA restrictions and non-determinism resolution language extensions respectively.

2.1 Restrictions on TIOA

The TIOA Simulator supports simulation of a subset of the TIOA language. We impose some restrictions on the trajectory definitions, quantifiers, and **for** loops. In particular, we restrict the form of **evolve** clauses and stopping conditions so that the simulator can automatically compute the values of the time-dependent variables that are reached after following a trajectory and detect any violation of the stopping conditions. We elaborate on these restrictions below.

2.1.1 Evolve clauses

TIOA allows a large spectrum of ways to express the evolution of time-dependent variables, such as arbitrary algebraic and differential equations. In a large class of the examples, time-dependent variables are used to model real time or clocks moving at the same rate as real time and possibly allowed to be reset. This observation, along with the fact that general differential equations are not always easy to automatically solve, motivated our decision in restricting the form of evolve clauses to only those of the form `d(var)=val`, where `var` is a time-dependent variable and `val` is the rate of evolution, as explained in detail below:

Time-dependent variables

The variable `var` can only be a variable reference, or a combination of the "select" (`variable.field`) or "get" (`variable[index]`) operators. In all cases, the variable referenced must be continuous and of type Real or AugmentedReal.

```
var     ::=  lvalue
lvalue  ::=  variable |
             ("."  <lvalue, operand>) |
             ("[]" <lvalue, operand+>)
```

As we discuss in Section 3-2, in order to follow a trajectory, the Simulator transforms stopping conditions to assignments of the form `var:=newval`, and for reasons of simplicity we require that `var` is also a valid l-value. Thus, the transformation to an assignment can occur automatically.

Rate of evolution

The rate of evolution `val` can be a literal term, a variable reference term or an application term that evaluates to a constant number. For example, the following **evolve** clauses are valid Simulator evolve clauses:

```
evolve d(now) = 1
% now is a state variable of type AugmentedReal

evolve d(timer) = u
% timer is a state variable of type Real
% u is an automaton formal parameter of type Int

evolve d(clock) = rates[i]
% now is a state variable of type AugmentedReal
% rates[i] is a constant number.
```

2.1.2 Stopping conditions

Ideally any predicate could be used in a stopping condition and the Simulator would be able to check that the stopping condition is not violated for any given amount of time. In general this would require to solve the stopping condition and evolution equations and find the earliest time where the stopping condition would hold. This would be the maximum point in time up to which the trajectory can be executed. Figure 2-1(left) shows an example of general evolution and stopping condition predicates. The solid line plots the time-dependent variable x as a function of time as described by an evolve predicate, and the dashed line plots x as a function of t as described in a stopping condition. The earliest (smallest t) intersection of the two lines illustrates the maximum point in time up to which the trajectory can be followed. In the general case, finding this point requires solving arbitrary equations for which exact solution methods might not be known. We therefore restrict the form of stopping conditions (in addition to the restriction of the evolve clauses):

Stopping conditions can be of the form `var=constant`.

Then, finding out the maximum point in time the trajectory can reach becomes easy, as shown in Figure 2-1(right). In fact, it is sufficient for the Simulator to check if the stopping conditions are violated at the end of a proposed (scheduled) time-passage event. We discuss this further in section 3.1.2. We now discuss how this restriction is relaxed.

It is often desired that a combination of boolean predicates on other state variables is also included in a stopping condition. Consider, for example a channel with time-bounded delivery guarantees. When it has no messages to deliver, time can advance forever. As soon as its queue becomes non-empty, time should not advance past the earliest deadline. Its stopping condition might then be:

```
stop when queue ≠ {} ∧ now = earliest(deadlines)
```

We can still check that stopping conditions of the above form hold since the extra predicates do not involve time-dependent variables. Moreover, if the automaton has more than one time-dependent variables, the stopping conditions should be allowed to check any subset of them, such as:

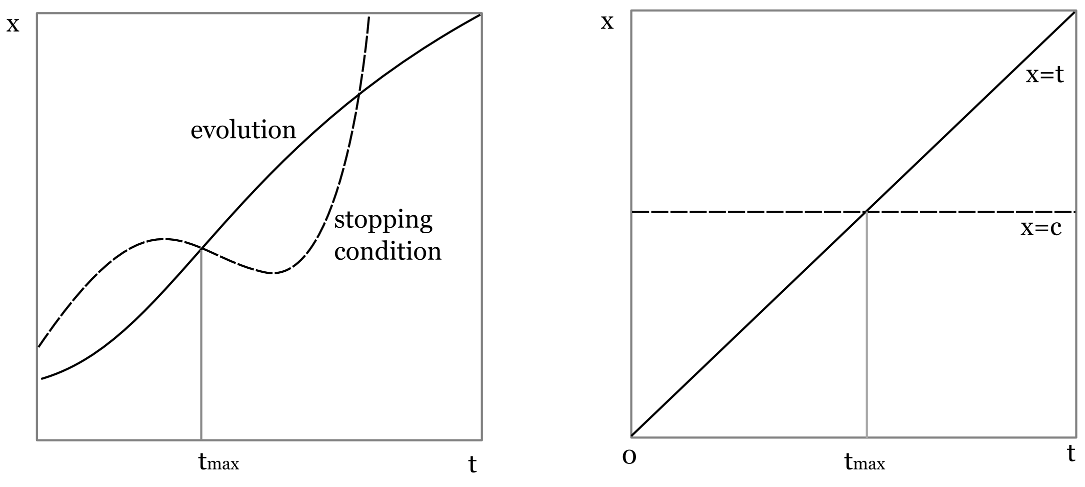


Figure 2-1: Left: In the general case we must solve the evolution and stopping condition equations to find the maximum point up to which the stopping condition is not violated (t_{max}). Right: The TIOA Simulator restricts the evolution to a linear equation and the stopping condition to a constant.

```
stop when clock1 = 5 ∨ clock2 = 39
```

However, we do *not* currently support directly comparing two time-dependent variables in the stopping conditions such as `stop when x =y + 1`, since this might require solving systems of equations. We formalize the restrictions on the stopping conditions in the following paragraphs.

Restrictions (formal)

The predicates in the stopping conditions are restricted as follows: A stopping condition can be a literal (`true` or `false`), a boolean variable reference term or an application term. If the literal or the variable evaluates to `true`, the trajectory will not be followed; if it evaluates to `false` the trajectory will be followed for any amount of time scheduled.

```
stopcondition ::= application:Bool | variable:Bool | literal:Bool
```

If the stopping condition is an application term, the following rules apply: Operands involving continuous variables can appear only in an equals (=) operator and only with a discrete operand to be compared with:

```
application ::= (operator, operand+) |
                ('=', <continuous_operand:Real, discrete_operand:constant:Real>) |
                ('=', <discrete_operand:constant:Real, continuous_operand:Real>)

operand ::= application | variable:discrete | literal

continuous_operand ::= lvalue:continuous

discrete_operand ::= (operator, <discrete_operand+>) | variable:discrete |
                     literal
```

2.1.3 Existential, universal quantifiers

No existential or universal quantifiers are allowed in the TIOA Simulator, unless the quantified variables are of type **enumeration**. This is because testing these quantifiers would require a way to enumerate all the possible values for a type, and there should only be a finite number of them. The only type that provides this for us is therefore that of **enumeration**.

Another exception is the `Nat` type, for which, even though an infinite type, we provide an enumeration for the first k natural numbers, where k is a certain finite constant. This exception allows useful quantified statements such as $\forall$ `n: Nat (n <len(queue)` $\Rightarrow$`queue[n] =0)`. The Simulator verifies quantifiers over natural numbers for the first k elements only, thus the guarantees of correct simulation with quantifiers over naturals are conditioned on the assumption that the value for k (which can be specified at runtime) is large enough to test all the relevant natural numbers of the quantifier.

2.1.4 Loops

The simulator allows **for** loops only if they are specified over finite sets, as in the example below.

```
% s : Set[Int]
% ok: Bool
for i: Int in s do
  if (i < 0) then ok := false fi
od
```

2.2 Non-determinism Resolution

As already indicated, TIOA inherits the non-deterministic nature of the mathematical framework and includes various sources of non-determinism, including the scheduling of transitions and trajectories and the explicit non-deterministic choices involving **choose** statements, **choose** parameters and **choose** expressions in initial assignments. Moreover, in order to be able to test simulation relations between two automata, we need a way of providing the simulator with a step correspondence.

The TIOA Simulator provides a mechanism for resolving non-determinism by letting the user explicitly specify which choice should be made at every point. This mechanism is an extension to the TIOA Language called the Non-Determinism Resolution language (NDR), and is derived from the NDR language used in IOA [3]. NDR can be used to schedule transitions and trajectories, resolve **choose** statements and specify the step correspondence for paired simulations. We provide an informal description of NDR in the following subsections, and a formal one in Appendix B.1

2.2.1 Scheduling transitions and trajectories

For the scheduling of transitions and trajectories the user must explicitly provide an execution schedule as an extension to an **automaton**. The schedule may contain its own state variables, specified by a **states** block, in the same way that the states of an **automaton** are specified. In a **do** ... **od** block, statements such as assignments, conditionals, **while** loops and statements to execute transitions and trajectories can be specified to drive the automaton's execution.

- Assignments and conditionals can be used as one would expect, with the exception that an automaton's state variables cannot be modified by the schedule block (and thus cannot appear on the left-hand side of the assignment).

- Instead of the TIOA **for** loops, NDR allows **while** loops. A **while** loop's program will be executed as long as its predicate evaluates to `true`.

- To execute a transition, the **fire** statement can be used. The statement requires the transition's type (input, output or internal), name and parameter values, if any.

- To execute a trajectory, a **follow** statement can be used. The statement should specify the trajectory's name, and the amount of time the trajectory should be followed.

- If the schedule block is in a composite automaton, the component's name should precede the state variables, transition, and trajectory names whenever used.

Figure 2-2 shows an example usage of an NDR schedule block for the automaton PeriodicSend in Figure 1-1 to resolve the scheduling of transitions and trajectories. In this particular example, the trajectory `traj` will be followed for a duration of `u` time units, and if the `clock` variable becomes equal to `u` (which should happen), the output transition `send` will be fired with the message `m1` as its parameter. This program is re-executed infinitely since it appears in a **while** `(true)` loop.

```
vocabulary Messages
  types M enumeration[nil, m1]

automaton PeriodicSend
  imports Messages
  signature
    output send(m: M)
  states
    u: Real := 5,
    clock: AugmentedReal := 0
  transitions
    output send(m)
      pre clock = u
      eff clock := 0
  trajectories
    trajdef traj
      stop when clock = u
      evolve d(clock) = 1

  schedule do
    while (true) do
      follow traj duration u;
      if (clock = u) then fire output send(m1) fi
    od
  od
```

Figure 2-2: Example of an NDR schedule for the automaton PeriodicSend

2.2.2 Explicit choose statement resolution

Explicit **choose** statements can be resolved by providing a deterministic program similar to the **schedule** block. The program is declared inside a **det do** ... **od** block, as in the example of Figure 2-3.

The simulator executes the NDR programs in a **choose** block, until a **yield** statement is encountered. Then the value of the **yield** statement is given to the variable making the

```
v := choose x where 0 ≤ x ≤ 10
               det do
                 yield 3; yield 6; yield randomInt(0,10);
               od;
```

Figure 2-3: Explicit choice statement resolution example

choice, in the above example, the variable `v`. If the block is executed again, the Simulator resumes execution from after the previous **yield**, starting over from the beginning if there are no statements left. In the example of Figure 2-3 therefore, the first time the block is executed the value of 3 will be chosen, the second one 6, the third one a randomly generated integer between 0 and 10 (or whatever the randomInt operator specifies), and so on.

2.2.3 Step correspondences

As already mentioned, the TIOA Simulator can be used to test simulation relations. For this purpose, the TIOA Simulator allows the user to specify a candidate simulation relation between two automata A and B, as well as a step correspondence that specifies:

- For each transition of the low-level automaton, the sequence of transitions that should be executed in the high-level one, and
- For each trajectory of the low-level automaton, the sequence of trajectories and internal transitions that should be executed in the high-level one.

Figure 2-4 shows an example of a simulation relation from an automaton `TCSpec` (Timed Channel Specification) to an automaton `TCImpl` (Implementation) which implements the specification using two queues. Messages are appended to the tail of the second queue and delivered from the head of the first queue. An internal `transfer` transition moves messages from the head of the second queue to the tail of the first.

Apart from the specifications of the two automata and the schedule block in the implementation one, the simulation relation is specified with a step correspondence inside the proof block. In this case the simulation relation is simply

```
TCSpec.queue = TCImpl.queue1 || TCImpl.queue2
```

where $\|$ is the operator for concatenation.

The step correspondence is also simple. External transitions and the trajectory are mapped to themselves, and the internal transition maps to the empty sequence.

```
automaton TCSpec(b: Real)
    where b ≥ 0
  imports Timestamp
  signature
    input send(m: M)
    output receive(m: M)
  states
    queue: Seq[TimedM] := {},
    now: AugmentedReal := 0
  transitions
    input send(m)
      eff queue := queue ⊢
                   [m, now+b]
    output receive(m)
      pre head(queue).message = m
      eff queue := tail(queue)
  trajectories
    trajdef traj
      stop when queue ≠ {} ∧
        now = head(queue).deadline
      evolve d(now) = 1
```

```
automaton TCImpl(b: Real)
    where b ≥ 0
  imports Timestamp
  signature
    input send(m: M)
    internal transfer(tm: TimedM)
    output receive(m: M)
  states
    queue1: Seq[TimedM] := {},
    queue2: Seq[TimedM] := {},
    now: AugmentedReal := 0
  transitions
    input send(m)
      eff queue2 := queue2 ⊢
                    [m, now+b]
    internal transfer(tm)
      pre head(queue2) = tm
      eff queue2 := tail(queue2);
          queue1 := queue1 ⊢ tm
    output receive(m)
      pre head(queue1).message = m
      eff queue1 := tail(queue1)
  trajectories
    trajdef traj
      stop when queue1 ≠ {} ∧
         now = head(queue1).deadline
      evolve d(now) = 1
```

```
  forward simulation from TCImpl to TCSpec:

    % The proposed simulation relation
    TCSpec.queue = TCImpl.queue1 ‖ TCImpl.queue2

    % The step correspondence
    proof
      for input send(m: M) do fire input send(m) od
      for internal transfer(tm: TimedM) ignore
      for output receive(m: M) do fire output receive(m) od
      for trajectory traj duration x do follow traj duration x od
```

Figure 2-4: A forward simulation from TimedChannelSpec to TimedChannelImpl

Chapter 3

TIOA Simulation

In the previous chapter we have discussed the conditions and extensions to TIOA that are necessary in order to allow simulation of Timed I/O Automata. This Chapter describes how simulation of TIOA is actually achieved, with a focus on the features that are new to TIOA, namely the time passage events. In particular, Sections 3.1, 3.2, and 3.3 discuss the design of the primitive automaton simulator, the composite automaton simulator, and the paired simulator respectively.

3.1 Primitive Automaton Simulation

The very first goal of the TIOA Simulator is to provide simulation of a single, primitive automaton. This section describes the various implementation issues in performing such a task, namely how the various TIOA data types are implemented, how the schedule block and other NDR statements are executed, how a transition is "fired" and how a trajectory is "followed", with particular focus given on the latter task, which is one of the major extensions we made to the IOA Simulator.

3.1.1 Data type implementations

The TIOA Simulator provides a large number of standard data types, ranging from Integer, Real, String, to more complex data types such as Map, Array, Sequence, Queue, Stack, Binary Search Tree, Enumeration, Union and Tuple. If the supplied data types are not sufficient, TIOA provides syntax for specifying new data types and operators (**vocabulary**), and the TIOA Simulator provides a way for the user to implement these data types in Java, and instruct the Simulator to find these implementations. Instructing the Simulator to find the data type implementations (what is called *registration* of data types) is exactly the same as it is for the IOA Simulator and IOA Compiler, as described in [13].

3.1.2 NDR execution

The Simulator executes the schedule blocks such as those of Figure 2-2 as one would expect, by going through the program and executing each statement. NDR conditionals, assignments and loops are executed as one would expect.

Firing transitions

For **fire** statements, we assign the given values to the transition parameters, check the preconditions of the transition, and if they hold execute the effect program of the transition. If the precondition fails, we terminate the execution providing an error message to the user.

Following trajectories

For **follow** statements, we first compute the final values of the time-dependent variables at the end of the trajectory based on the **follow** statement's duration and the evolve clauses. We then check that the stopping conditions will not be violated with these values and that the invariants of the trajectory will hold with both the initial and the final values of the time-dependent variables. If none of the stopping conditions and invariants are violated, the final values are assigned to the time-dependent variables and execution resumes in the schedule block; otherwise, we halt with an error.

Note that checking that the invariants hold only at the beginning and at the end of the time-passage event does not guarantee that the invariant holds throughout the event. In general, it is impossible to guarantee this unless we restrict the form of the invariants. Instead, we allow arbitrary invariants and draw the user's attention that the invariants are tested only at the beginning and at the end of each time-passage event. For simple invariants of the type `varterm op valterm` where `varterm` evaluates to a time-dependent variable, `valterm` to a discrete variable and `op` is a comparison operator such as $<$, $\leq$, $=$, $\geq$, $>$, testing only the beginning and the end of the event actually guarantees that the invariant holds.

We now describe the execution of trajectories in detail. Whenever a **follow** statement is encountered in the execution of the schedule, the TIOA Simulator translates the trajectory definition to a program, as shown in Figure 3-1. Then, this program is executed as a normal TIOA program. Before advancing time (assigning the new values to the time-dependent variables), we check that the invariant of the trajectory holds. If the invariant does not hold then we halt the execution with an error message. Otherwise, we then compute the values of the time-dependent variables after the time-passage event as we discuss below and assign those values to the time-dependent variables. Finally, we evaluate the invariant once again and the stopping conditions. If any of them fail, once again the execution of this trajectory is an error.

```
if (¬invariant) then error fi;

var1 := newValue1;
var2 := newValue2;
% ... for all k time-dependent variables
vark := newValuek;

if (¬invariant ∨ stopCondFails) then error fi;
```

Figure 3-1: Pseudocode showing the program executed for each follow statement. The `error` statement instructs the simulator to halt execution with an error message.

As Figure 3-1 shows, the translation of a **follow** statement to such a program involves finding the values of the time-dependent variables at the end of the time-passage event (the `newValuei` terms) given the variable, rate, and duration rate, and the `stopCondFails` predicate given the stopping condition, the time-dependent variables and their rates.

Values of variables at the end of the trajectory

Given a time-dependent variable *var*, the rate at which the variable evolves *rate* and the duration of the time passage event given in the **follow** statement, *duration*, we want to compute the value that would result at the end of the time-passage event. Given that the Simulator only allows evolve clauses of the form `d(var)=val`, the new value is given by the formula:

$$var + (rate * duration)$$

The stopping condition failure predicate

Given a stopping condition predicate from a trajectory's definition, the set of the time-dependent variables of the automaton and a mapping from these variables to the rate with which they are evolving (from the `evolve` clause), we generate a predicate on the variables of the stopping condition that is true if the stopping condition would be violated for given values for the variables. Figure 3-2 specifies in pseudocode the `convert` procedure, which given a stopping condition `t` returns a new predicate that is identical to `t`, with the exception of `var =value` and `value =var` terms. In these cases, the = operator is converted to > if the rate at which `var` grows is non-negative, and to < if the rate is negative.

```
convert(ApplicationTerm t, Set[Var] timeDependVars, Map[Var, Real] rates)
  if (t.operator.name = "=" ∧
      both t.operands ∈ timeDependVars) then
        error
  elseif (t.operator.name = "=" ∧
          only 1 of t.operands ∈ timeDependVars) then
        let var be the operand that ∈ timeDependVars,
        rate = rates[var],
        value be the other operand :
    return a new ApplicationTerm t′ with:
            t′.operands[0] = var
            t′.operands[1] = value
            t′.operator = ">" if rate ≥ 0, "<" otherwise
  else
    let operand0 = convert(t.operands[0], timeDependVars, rates),
        ...
        operandk = convert(t.operands[k], timeDependVars, rates) :
    return a new ApplicationTerm t′ with
        t′.operator = t.operator,
        t′.operands = {operand0, ... operandk}
```

Figure 3-2: Converting the stopping condition to a stopping condition failure predicate

3.2 Composite Automaton Simulation

Motivation

The ability to simulate a system that consists of more than one component is useful in the process of evaluating the correctness, fault tolerance, and availability of a system. The TIOA Simulator should therefore provide a way in which composite automata can be simulated. One possibility is to require users to "expand" a composite automaton, either manually or by using an automatic tool, so that the automaton becomes a primitive one that encapsulates its components in its state. Joshua A. Tauber demonstrates an automatic tool for expanding composite IOA automata[12, 11]; an extension of that tool could be used for this purpose, for example. Simulation of this expanded automaton is possible by providing an NDR schedule block for the automaton and using the primitive TIOA Simulator. On the other hand, this process has some drawbacks. First, the ability to provide a schedule block for every component independently is ruled out. As we discuss below, this option can be very helpful. Second, even automatic expansion is sometimes hard to get right, and its semantics for the combination of transitions with different **where** clauses are hard to specify and use. Finally, fixing a bug in the expanded automaton would also require to manually trace the bug back in the individual components and perform the change there as well. Overall, the ability to simulate composite automata without requiring the user to expand the automata first is very useful.

Scheduling

Similar to the IOA Simulator[10], the TIOA Simulator provides two alternative options to simulate composite automata:

- **Option 1:** The user may provide a single schedule block for the composition, and no schedule blocks for individual components. This option can be useful if it is easier to reason about a system as a whole, or if a specific "test case" execution for system is desirable.

- **Option 2:** The user can provide a schedule block for each individual component, and no schedule block for the composition. This option can be useful if it is easier to specify the schedules of the independent components rather than that of the whole system. Moreover, the user might already have schedule blocks for components that they have already written and tested during primitive simulation. Reusing these schedules is therefore desirable. If this option is used, the simulator will give turns to the components in a random, weighted random or deterministic way, thus this method can be used to test the system over multiple “test case” executions.

3.2.1 Schedule block for the composition

This option allows the user to provide a single schedule block for the composition. The simulator attempts to execute the schedule block similar to the execution of primitive automaton schedule blocks. The framework specifies that if one component allows time passage for a specific amount of time, then so must all other components of the system. Thus, the NDR allows simultaneous **follow** statements in composite schedule blocks, as shown below:

```
follow A.traj, B.traj, C.traj duration 10
```

Whenever a **follow** statement is encountered in the composition’s schedule block, the simulator attempts to execute the trajectories (as with primitive simulation by checking for any violations of the stopping conditions after the time passage or of the invariant before and after the time passage).

3.2.2 Schedule block for the components

Another way to achieve simulation of a composite automaton is to specify a schedule block for each individual component of the system, instead of the composition itself. The simulator gives turns to the components (in either a random, weighted random or uniform way), executing their schedules.

Connected actions

One important difference in this case is that connected actions (output and input actions with the same name) among different components must be fired at the same time. Thus, for every output action the Simulator is about to execute in one component, it looks for the set of corresponding input actions in other components and fires them as well.

Scheduling inputs

Input actions are allowed to be scheduled, and the TIOA Simulator acts differently depending on whether the action has a corresponding output one in another component. If it does not, then the action is executed normally. If it does, then the input action is simply ignored, because it will be executed automatically when the corresponding output action will be scheduled.

Simultaneous time passage events

As with the previous option, we must ensure that all components follow their time-passage events simultaneously. We achieve this by pausing the execution of components that reach a **follow** statement and give turns to the other components, until all the components are ready to follow a trajectory. We then follow all the trajectories together for the maximum duration possible, and update the schedules accordingly. For components whose trajectories were followed completely, we move on to the next statement in their schedule, and for the rest we indicate the amount of time still left to follow. The algorithm is described in detail below:

Composite simulation (schedules in the components) algorithm

For every component C of the system, we maintain two variables: *trajectory-waiting*, a flag, indicating whether or not the component is ready to follow a trajectory, and *duration*, an AugmentedReal, indicating the amount of time that must pass for the component to move on to the next statement in its schedule block. The algorithm then executes as follows: For each component C in the system, chosen either at random or in a round-robin way (as the user may specify): Let s be the next statement in its schedule block. Then,

- If s is a conditional, loop, or assignment statement, execute s normally.
- If s is a **fire** statement, execute s normally and if the transition is an output one, fire the corresponding input ones as well, and
- If s is a **follow** statement, then: Pause the execution of C; Indicate that C is in a *trajectory-waiting* status with a *duration* left equal to the one indicated by the statement's **duration** keyword. Let NW be the set of all the system's components

that are not in a *trajectory-waiting* status. If NW is not empty, this means that there is at least one (other) component not waiting for a trajectory. We then exit C's execution and yield the turn to one of the components in NW. Otherwise, all components are waiting for their trajectories. Then, let d be the maximum duration that all components can follow without violating their stopping conditions, that is, the minimum of all components' *duration* variable. Follow the trajectories of all components for d time units. Subtract d from all components' *duration* variables. For each component whose *duration* becomes 0, set *trajectory-waiting* to `false` and move their program counter to the next statement.

3.3 Paired Simulation

The TIOA Simulator allows the user to specify a candidate simulation relation between two automata A and B, as well as a step correspondence, of the form:

- For each transition `a` of the low-level automaton, execute a sequence of transitions of the high-level one.
- For each trajectory `t` of the low-level automaton, execute a sequence of trajectories and internal transitions of the high-level one.

The Simulator executes the automata together, checks that the simulation relation holds and that the external behavior of the two automata is the same. A schedule block in the low-level automaton drives the execution. For each transition and trajectory about to be executed in the low-level automaton, the corresponding sequence of transitions and trajectories is found (with the help of the step correspondence) and attempted to be executed in the high-level automaton as well. The simulator verifies that the external behavior of the two automata is the same, that the simulation relation holds initially and after every step taken, and that the invariants of both automata are not violated initially and after every step taken.

The TIOA Simulator currently supports paired simulations of primitive automata only. However, it should be easy to extend this to support composite automata as well.

Chapter 4

Case Study: Failure Detection

We provide a simple example of a distributed system with timing guarantees that has been specified, simulated, and proved correct using the TIOA tools. The example is the failure detection system from [7]. The system consists of three components:

- A sending process (P) that sends a message every $u1$ time units and has the potential of coming to a stopping failure,
- A channel (C) that delivers all its messages reliably within a time bound of b time units, and
- A timeout process (T) that detects the failure of the sending process by timing out. The timeout process indicates that a failure has occurred in P if $u2 > u1 + b$ time has passed since it last received a message from P.

In Section 4.1 we specify the primitive automata for the components of the system, provide sample NDR schedule blocks for each component, and show the output of the TIOA Simulator for these schedules. In Section 4.2 we specify the No Failure and Failure Detection systems using a composition of the individual components, and illustrate the two different options in simulation of composite automata: schedules in the components or schedule in the composition. Simulator traces for both systems using both options are also shown. In Section 4.3 we show a paired simulation between two primitive automata, the hand-composed Failure Detection system's implementation and the system's specification. The NDR schedule in the implementation system and the provided step correspondence drive the paired simulation.

4.1 Simulating Primitive Automata

4.1.1 Periodic send

In Figure 4-1, the `PeriodicSend` automaton uses the continuous variable `clock` as a timer to send a message every `u` time units. When a `send(m)` transition occurs, the timer is reset and another send cannot occur until `clock` =`u`. Its trajectory `traj` must stop when it is time to send a new message. The provided NDR schedule is a simple infinite loop that follows `traj` for `u` time units and fires the output transition `send` with a message (`m1`). Simulation of the `PeriodicSend` automaton with that schedule and the value 5 for the formal parameter $u1$ results in the trace shown in Figure 4-1. The trace repeats itself every two steps. The actual values for the formal parameters for this and subsequent automata are loaded from a file with contents shown below (For the syntax of the formals file, see Appendix A.2.1

```
((u1 tioa.runtime.adt.RealSort 5)
 (b  tioa.runtime.adt.RealSort 2)
 (u2 tioa.runtime.adt.RealSort 8))
```

```
vocabulary Messages
  types M enumeration[nil, m1]

automaton PeriodicSend(u1: Real)
    where u1 > 0
  imports Messages
  signature
    output send(m: M)
  states
    clock: AugmentedReal := 0
  transitions
    output send(m)
      pre clock = u1
      eff clock := 0
  trajectories
    trajdef traj
      stop when clock = u1
      evolve d(clock) = 1
  schedule do
    while (true) do
      follow traj duration u1;
      fire output send(m1)
    od
  od
```

```
Automaton initialized
1:   trajectory traj for 5.0 units
2:   output transition send(m1)
3:   trajectory traj for 5.0 units
4:   output transition send(m1)
...
```

Figure 4-1: Periodic Send with no failure

4.1.2 Periodic send with failure

In Figure 4-2 the `PeriodicSend2` automaton is specified, which is a modification of `PeriodicSend` that allows for a stopping failure to occur. The failure is modeled with an input transition (`fail`) which sets the `failed` flag. This disables the `send` transition and allows the `traj` trajectory to be followed for an infinite amount of time. In our sample NDR scheduler, we send two rounds of messages before failing. After failure we follow the trajectory for `\infty` time units. The trace for this execution, using $u1 = 5$ is also shown in Figure 4-2.

```
vocabulary Messages
  types M enumeration[nil, m1]

automaton PeriodicSend2(u1: Real)
    where u1 > 0
  imports Messages
  signature
     input fail
     output send(m: M)
  states
     failed: Bool := false,
     clock: AugmentedReal := 0
  transitions
     output send(m)
       pre ¬failed ∧ clock = u1
       eff clock := 0
     input fail
       eff failed:= true
```

```
trajectories
  trajdef traj
    stop when
      ¬failed ∧ u1 = clock
    evolve
      d(clock) = 1
schedule
  states
    count: Nat := 0,
    n: Nat := 2
  do
    % Send n rounds of messages
    while (count < n) do
      follow traj duration u1;
      fire output send(m1);
      count := count + 1
    od;
    fire input fail;
    follow traj duration \infty
  od
```

```
Automaton initialized
1:   trajectory traj for 5.0 units
2:   output transition send(m1)
3:   trajectory traj for 5.0 units
4:   output transition send(m1)
5:   input transition fail
6:   trajectory traj for Infinity units
No more steps
No errors
```

Figure 4-2: Periodic Send with failure

4.1.3 Reliable channel with deadline guarantees

In Figure 4-3 we model a reliable channel that ensures delivery of its messages within b time units of their receipt. We first specify the `TimedM` type that augments a message with deadline. The channel enqueues messages it receives through the `send` input action in a queue, and sets their deadline to `now + b`. Its trajectory may be followed for any amount of time when the queue is empty, otherwise it should stop before or exactly at the time of the first message's deadline. Since all messages have the same maximum delay, the deadlines in the queue are monotonically non-decreasing thus the first element in the queue always has the earliest deadline.

In our sample scheduler, in every phase of the execution (every loop), we randomly decide whether or not to send a message to the channel. Then, if the queue is empty we follow the trajectory for some amount of time that is less than b (specifically, we chose $b/2$). Otherwise, we follow the trajectory up to the point where the the first message's deadline would be met and deliver the message. Another possible schedule could deliver the message earlier instead of waiting until its deadline. The schedule and a sample execution are shown in Figure 4-3.

4.1.4 Failure detector

The final component of our system is the process that receives the messages and detects any failures. The `Timeout` automaton of Figure 4-4 maintains a flag called `suspected` that indicates whether or not the sending process is suspected to have failed. This becomes true only when $u2$ time units have passed without receiving a message. Similar to `PeriodicSend`, the `clock` variable is used as a timer that is reset every time a message is received. The automaton's trajectory may be followed for any time duration when the process is suspected, but it should stop if the timer reaches $u2$, so that the `timeout` action can occur.

The provided schedule block randomly decides whether to receive a message in every round. It then checks whether it has not received a message for the past $u2$ units, in which case it fires a `timeout` action, otherwise it allows $u2/2$ time to pass before checking again. If the sending process is already suspected of having failed, it allows an infinite amount of time to pass. Every execution of this scheduler should result in different traces because of the `randomBool` operator. We show one with $u2 = 8$ and where a message was not received in rounds 1,2,5,6 and 7, and thus a timeout occurred in the seventh round.

```
vocabulary Messages
  types M enumeration[nil, m1]
vocabulary Timestamp
  imports Messages
  types TimedM tuple [message: M, deadline: AugmentedReal]
vocabulary Random
  operators randomBool: → Bool
```

```
automaton TimedChannel(b: Real)
     where b ≥ 0
  imports Timestamp, Random
  signature
    input send(m: M)
    output receive(m: M)
  states
    queue: Seq[TimedM] := {},
    now: AugmentedReal := 0
  transitions
    input send(m)
      eff queue := queue ⊢
                     [m, now+b]
    output receive(m)
      pre head(queue).message = m
      eff queue := tail(queue)
```

```
trajectories
  trajdef traj
    stop when queue ≠ {} ∧
       now = head(queue).deadline
    evolve d(now) = 1
schedule do
    while (true) do
      if randomBool = true then
         fire input send(m1)
      fi;
      if queue = {} then
        follow traj duration b/2
      else
        follow traj duration
          head(queue).deadline - now;
        fire output
          receive(head(queue).message)
      fi
    od
  od
```

```
Automaton initialized
1:   trajectory traj for 1.0 unit
2:   trajectory traj for 1.0 unit
3:   trajectory traj for 1.0 unit
4:   input transition send(m1)
5:   trajectory traj for 2.0 units
6:   output transition receive(m1)
7:   trajectory traj for 1.0 unit
8:   input transition send(m1)
9:   trajectory traj for 2.0 units
10:  output transition receive(m1)
...
```

Figure 4-3: Reliable Channel with deadline guarantees

```
vocabulary Messages
  types M enumeration[nil, m1]
vocabulary Random
  operators randomBool: → Bool

automaton Timeout(u2: Real)
     where u2 > 0
  imports Messages, Random
  signature
    input receive(m: M)
    output timeout
  states
    suspected: Bool := false,
    clock: AugmentedReal := 0
  transitions
    input receive(m)
      eff clock:=0;
          suspected:= false
    output timeout
      pre ¬suspected ∧ clock = u2
      eff suspected := true
```

```
trajectories
  trajdef traj
    stop when
      ¬suspected ∧ clock = u2
    evolve
      d(clock) = 1
schedule
  states done : Bool := false
do while (¬done) do
  if (¬suspected) then
    if randomBool then
      fire input receive(m1)
    fi;
    if clock = u2 then
      fire output timeout
    else
      follow traj duration u2/2
    fi
  else
    follow traj duration \infty;
    done := true
  fi
od
od
```

```
Automaton initialized
1:   trajectory traj for 4.0 units
2:   trajectory traj for 4.0 units
3:   input transition receive(m1)
4:   trajectory traj for 4.0 units
5:   input transition receive(m1)
6:   trajectory traj for 4.0 units
7:   trajectory traj for 4.0 units
8:   output transition timeout
9:   trajectory traj for Infinity units
No more steps
No errors
```

Figure 4-4: Timeout

4.2 Simulating Composite Automata

In the previous section we specified and tested all the components of the system independently. Testing the system as a whole and the interactions of the components is not possible unless we perform a composite simulation. In Section 4.2.1 we show the first option in simulating a composite system, which is to include the schedules for the individual components and not for the composition. Alternatively, we can test the system by providing a schedule in the composition and not in the components, as we do in Section 4.2.2. For each option, we test two systems: The `No Failure` system where the `PeriodicSend` process does not fail, and the `Failure Detection` system in which the sending process fails.

4.2.1 Schedules in the components

No failure

In Figure 4-5 we provide a composition of one instance of `PeriodicSend`, `TimedChannel` and `Timeout` automata. The file in which the system is specified also includes the specifications and NDR schedule blocks of `PeriodicSend`, `TimedChannel` and `Timeout` shown in Figures 4-1, 4-3 and 4-4. The `Composition` automaton simply specifies one instance of each component and provides values for their formal parameters.

Simulation of the system with $u1 = 5, b = 2, u2 = 8$ results in the trace shown in Figure 4-5. After 5 time units, the component `P` sends a message. The input action `send` of the `C` component is also fired at the same time. After 2 time units the channel delivers the message to `T`, and 3 units later the process starts over again. The trajectories are some times broken into 1-unit steps since the `TimedChannel` process follows its trajectory every $b/2$ units when its queue is empty.

Failure detection

The composite automaton of Figure 4-6 is identical to that of Figure 4-5 except from the fact that it uses `PeriodicSend2` which can fail. The file in which the system is specified also includes the specifications and NDR schedule blocks of `PeriodicSend2`, `TimedChannel` and `Timeout` shown in Figures 4-2, 4-3 and 4-4. The `Composition` automaton simply specifies one instance of each component and provides values for their formal parameters.

Simulation of the system with $u1 = 5, b = 2, u2 = 8$ results in the trace shown in

```
% specifications and schedules of PeriodicSend, TimedChannel and Timeout
% ...

automaton Composition(u1, u2, b: Real) where (u1 + b) < u2
  components
    P: PeriodicSend(u1);
    C: TimedChannel(b);
    T: Timeout(u2);


Automaton initialized
1:   trajectory T.traj, C.traj, P.traj for 1.0 unit
2:   trajectory T.traj, C.traj, P.traj for 1.0 unit
3:   trajectory T.traj, C.traj, P.traj for 1.0 unit
4:   trajectory T.traj, C.traj, P.traj for 1.0 unit
5:   trajectory T.traj, C.traj, P.traj for 1.0 unit
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory T.traj, C.traj, P.traj for 1.0 unit
8:   trajectory T.traj, C.traj, P.traj for 1.0 unit
9:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
10:  trajectory T.traj, C.traj, P.traj for 1.0 unit
11:  trajectory T.traj, C.traj, P.traj for 1.0 unit
12:  trajectory T.traj, C.traj, P.traj for 1.0 unit
13:  output transition P.send(m1), connected to:
     input transition C.send(m1)
14:  trajectory T.traj, C.traj, P.traj for 2.0 units
15:  output transition C.receive(m1), connected to:
     input transition T.receive(m1)
...
```

Figure 4-5: No Failure System

Figure 4-6. After 5 time units, the component `P` sends a message through the channel. After 2 time units the channel delivers the message to `T`. A message is sent once more, as the schedule of `PeriodicSend2` specifies, and at that point `P` fails (step 14). The message is delivered and 8 ($u2$) time units after the delivery `T` times out. From then on, no actions are enabled and the trajectories of the components are followed for an infinite amount of time, broken into 1-unit steps. This break happens since the `TimedChannel` process follows its trajectory every $b/2$ units when its queue is empty.

4.2.2 Schedule in the composition

An alternative to providing shadeless in the individual components is to write a schedule for the composite automaton itself. We discuss the same examples (`NoFailure` and `FailureDetection`) scheduled in this way.

No failure

The `Composition` automaton of Figure 4-7 is identical to that of Figure 4-5, but includes an NDR schedule. The file also includes the specifications of `PeriodicSend`, `TimedChannel` and `Timeout`, but without their schedule blocks.

The schedule we provide enters an infinite loop in which every $u1$ units `P` sends a message that is delivered b units later, as the trace with $u1 = 5, b = 2, u2 = 8$ verifies.

Failure detection

The system of Figure 4-8 composes `PeriodicSend2` with the channel and timeout processes, and includes an NDR schedule in the composition. The file must also includes the specifications (without the schedule blocks) of `PeriodicSend2`, `TimedChannel` and `Timeout`.

The schedule we provide specifies that `P` sends $n = 2$ messages before failing. After its failure and the delivery of its last message, it is detected and a `timeout` action occurs. The trace with $u1 = 5, b = 2, u2 = 8$ verifies this behavior.

```
% specifications and schedules of PeriodicSend, TimedChannel and Timeout
% ...

automaton Composition(u1, u2, b: Real) where (u1 + b) < u2
  components
    P: PeriodicSend2(u1);
    C: TimedChannel(b);
    T: Timeout(u2);


Automaton initialized
1:   trajectory P.traj, T.traj, C.traj for 1.0 unit
2:   trajectory P.traj, T.traj, C.traj for 1.0 unit
3:   trajectory P.traj, T.traj, C.traj for 1.0 unit
4:   trajectory P.traj, T.traj, C.traj for 1.0 unit
5:   trajectory P.traj, T.traj, C.traj for 1.0 unit
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory P.traj, T.traj, C.traj for 1.0 unit
8:   trajectory P.traj, T.traj, C.traj for 1.0 unit
9:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
10:  trajectory P.traj, T.traj, C.traj for 1.0 unit
11:  trajectory P.traj, T.traj, C.traj for 1.0 unit
12:  trajectory P.traj, T.traj, C.traj for 1.0 unit
13:  output transition P.send(m1), connected to:
     input transition C.send(m1)
14:  input transition P.fail
15:  trajectory P.traj, T.traj, C.traj for 1.0 unit
16:  trajectory P.traj, T.traj, C.traj for 1.0 unit
17:  output transition C.receive(m1), connected to:
     input transition T.receive(m1)
18:  trajectory P.traj, T.traj, C.traj for 1.0 unit
19:  trajectory P.traj, T.traj, C.traj for 1.0 unit
20:  trajectory P.traj, T.traj, C.traj for 1.0 unit
21:  trajectory P.traj, T.traj, C.traj for 1.0 unit
22:  trajectory P.traj, T.traj, C.traj for 1.0 unit
23:  trajectory P.traj, T.traj, C.traj for 1.0 unit
24:  trajectory P.traj, T.traj, C.traj for 1.0 unit
25:  trajectory P.traj, T.traj, C.traj for 1.0 unit
26:  output transition T.timeout
27:  trajectory P.traj, T.traj, C.traj for 1.0 unit
...
```

Figure 4-6: Failure Detection System

```
% specifications (without schedules) of PeriodicSend, TimedChannel and Timeout
% ...

automaton Composition(u1, u2, b: Real) where (u1 + b) < u2
  components
    P: PeriodicSend(u1);
    C: TimedChannel(b);
    T: Timeout(u2);
schedule do
  follow P.traj, C.traj, T.traj duration u1;
  while (true) do
    fire output P.send(m1);
    follow P.traj, C.traj, T.traj duration b;
    fire output C.receive(m1);
    follow P.traj, C.traj, T.traj duration (u1-b);
  od
od
```

```
Automaton initialized
1:   trajectory P.traj, C.traj, T.traj for 5.0 units
2:   output transition P.send(m1), connected to:
     input transition C.send(m1)
3:   trajectory P.traj, C.traj, T.traj for 2.0 units
4:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
5:   trajectory P.traj, C.traj, T.traj for 3.0 units
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory P.traj, C.traj, T.traj for 2.0 units
8:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
...
```

Figure 4-7: No Failure System with schedule in the composition

```
% specifications (without schedules) of PeriodicSend2, TimedChannel and Timeout
% ...

automaton Composition(u1, u2, b: Real) where (u1 + b) < u2
  components
    P: PeriodicSend2(u1);
    C: TimedChannel(b);
    T: Timeout(u2);
schedule
  states
    count: Nat := 0,
    n: Nat :=2
  do
    follow P.traj, C.traj,
      T.traj duration u1;
    % Send n messages before failing
    while (count < n) do
      fire output P.send(m1);
      follow P.traj, C.traj, T.traj duration b;
      fire output C.receive(m1);
      follow P.traj, C.traj, T.traj duration (u1-b);
      count := count + 1
    od;
    % failure
    fire input P.fail;
    follow P.traj, C.traj, T.traj duration u2 - (u1-b);
    % detection
    fire output T.timeout;
    follow P.traj, C.traj, T.traj duration \infty;
  od
```

```
Automaton initialized
1:   trajectory P.traj, C.traj, T.traj for 5.0 units
2:   output transition P.send(m1), connected to:
     input transition C.send(m1)
3:   trajectory P.traj, C.traj, T.traj for 2.0 units
4:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
5:   trajectory P.traj, C.traj, T.traj for 3.0 units
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory P.traj, C.traj, T.traj for 2.0 units
8:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
9:   trajectory P.traj, C.traj, T.traj for 3.0 units
10:  input transition P.fail
11:  trajectory P.traj, C.traj, T.traj for 5.0 units
12:  output transition T.timeout
13:  trajectory P.traj, C.traj, T.traj for Infinity units
No more steps
No errors
```

Figure 4-8: Failure Detection System with schedule in the composition

4.3 Paired Simulation

Paired simulations enable testing of simulations relation which indicate the relationship between the states of an implementation and a specification. If a simulation relation is proved, the implementation system is then shown to satisfy the specifications and its properties. Proving a simulation relation usually requires showing for each step of the implementation starting from an implementation state that is related to a specification state, which sequence of steps should be taken by the specification system to result in a new state that is also related to the implementations's new state.

Both the simulation relation and its proof steps are not always easy to come up with, and are certainly very hard for a program to discover them automatically. They must therefore be provided to the Paired Simulator. In the following subsections we show an example of a system's specification, implementation, and a simulation relation. The system is the Failure Detection system which has already been implemented and simulated in the previous sections.

4.3.1 Failure detection specification

In Figure 4-9 we provide an abstract specification of the failure detection system. The system is specified as a single process that might `fail` and `timeout`. It keeps track of two flags, `suspected` and `failed` that carry the same meaning as in the implementation system. The `last_timeout` variable indicates the latest time a `timeout` transition should occur, and `now` grows at the same rate as real time. When a failure occurs, we set `fail` to true and `last_timeout` to `now + u2 + b` and when a timeout occurs we set `last_timeout` to `\infty` and `suspected` to `true`. The trajectory must stop if a failure has occurred, a timeout has not occurred and `now` has reached the `last_timeout`.

4.3.2 Failure detection implementation

Figure 4-10 provides the implementation of the Failure Detection system, in an "expanded" composition form. This means that we have transformed our composition automation into a primitive one by: (a) encapsulating the state of each component in the state of the composition (b) merging transitions by conjunctions of the preconditions and composition of the effect programs and by (c) merging the trajectory definitions by disjunctions of

```
automaton FDSpec(u1,u2,b:Real)
    where (u1 + b) < u2
  signature
    internal fail
    output timeout
  states
    last_timeout:
      AugmentedReal := \infty,
    now : AugmentedReal := 0,
    suspected: Bool := false,
    failed: Bool := false
  transitions
    internal fail
      pre ¬failed
      eff failed := true;
          last_timeout :=
             now + u2 + b
    output timeout
      pre failed ∧ ¬suspected
      eff suspected := true;
          last_timeout := \infty
```

```
trajectories
  trajdef traj
    stop when
       failed ∧
       ¬suspected ∧
       now = last_timeout
    evolve
      d(now) = 1

invariant S of FDSpec:
  now ≥ 0;
  suspected ⇒ failed;
  failed ∧ ¬suspected ⇔
    \infty ≠ last_timeout;
  now ≥ 0 ⇒ now ≤ last_timeout;
  (now + u2 + b) ≥ 0 ∧
  \infty ≠ last_timeout ⇒
     last_timeout ≤ (now + u2 + b)
```

Figure 4-9: Failure Detection System Specification

the stopping conditions and compositions of the evolve classes. This step was necessary because the current version of the TIOA simulator does not support paired simulations among composite automata.

The implementation system is also accompanied by a schedule that will drive the execution of both systems during the paired simulation. This is identical to the schedule in Figure 4-8. The invariant of the specific implementation is also provided.

4.3.3 Forward simulation

The relation among the states and the step correspondence can now be specified and tested. The relation itself is a set of predicates relating the states of the implementation and the specification. The step correspondence is provided in a **proof** block (the name implies the fact that an actual proof would specify these step correspondences as well). Providing the implementation automation and schedule, specification automation and forward simulation with the step correspondence as those of Figure 4-11 in a file allows us to perform a paired simulation. A trace from the paired simulation with $u1 = 5, b = 2, u2 = 8$ is shown in Fig 4-11.

```
vocabulary Composition
  types M enumeration[nil, m1]
        TimedM tuple [message: M, timestamp: AugmentedReal]
        PeriodicSend2 tuple [failed: Bool, clock: AugmentedReal]
        TimedChannel  tuple [queue: Seq[TimedM], now: AugmentedReal]
        Timeout       tuple [suspected: Bool, clock: AugmentedReal]
```

```
automaton FDImpl(u1,u2,b:Real)
    where (u1 + b) < u2
  imports Composition
  signature
    internal fail
    internal send(m: M)
    internal receive(m: M)
    output timeout
  states
    P: PeriodicSend2 := [false, 0],
    C: TimedChannel  := [{}, 0],
    T: Timeout       := [false, 0]
  transitions
    internal send(m)
      pre ¬P.failed ∧ P.clock = u1
      eff P.clock := 0;
          C.queue :=
            C.queue ⊢ [m, C.now + b]
    internal fail
      eff P.failed:= true
    internal receive(m)
      pre head(C.queue).message = m
      eff C.queue := tail(C.queue);
          T.clock:=0;
          T.suspected:= false
    output timeout
      pre ¬T.suspected ∧
           T.clock = u2
      eff T.suspected := true
  trajectories
    trajdef traj
      stop when
        (C.queue ≠ {} ∧
         head(C.queue).timestamp =
           C.now) ∨
        (¬T.suspected ∧
           T.clock = u2) ∨
        (¬P.failed ∧ P.clock = u1)
      evolve
        d(P.clock) = 1;
        d(C.now) = 1;
        d(T.clock) = 1
```

```
schedule
  states
    count: Nat := 0,
    n: Nat :=2
  do
    follow traj duration u1;
    % Send n rounds of messages
    while (count < n) do
      fire internal send(m1);
      follow traj duration b;
      fire internal receive(m1);
      follow traj duration (u1-b);
      count := count + 1
    od;
    % failure
    fire internal fail;
    follow traj duration u2 - (u1-b);
    % detection
    fire output timeout;
    follow traj duration \infty
  od

invariant I of FDImpl:
  C.now ≥ 0;
  C.now ≥ 0 ∧ C.queue ≠ {} ⇒
    C.now ≤
      (head(C.queue)).timestamp;
  (C.now + u2) ≥ 0 ∧ ¬T.suspected ⇒
    T.clock ≠ \infty ∧ T.clock ≤ u2;
  (C.now + u1) ≥ 0 ∧ ¬P.failed ⇒
    P.clock ≠ \infty ∧ P.clock ≤ u1;
  ∀ n: Nat (n < len(C.queue) ⇒
    C.queue[n].timestamp ≤
      (C.now + b));
  b ≥ 0 ∧ ¬ P.failed ⇒
    (if C.queue ≠ {}
     then (head(C.queue)).timestamp <
        (T.clock + (C.now + u2))
     else (P.clock + b) <
        (T.clock + (C.now + u2)));
  T.suspected ⇒ P.failed
```

Figure 4-10: Failure Detection System Implementation

```
forward simulation from FDImpl to FDSpec:

  % Simulation Relation
  FDImpl.P.failed = FDSpec.failed;
  FDImpl.T.suspected = FDSpec.suspected;
  FDImpl.C.now = FDSpec.now;
  (¬FDSpec.failed ⇒ FDSpec.last_timeout = \infty);
  ((FDSpec.failed ∧ FDImpl.C.queue ≠ {}) ⇒
    ∀ k: Nat (k < len(FDImpl.C.queue) ⇒
      FDSpec.last_timeout ≥ FDImpl.C.queue[k].timestamp));
  ((FDSpec.failed ∧ FDImpl.C.queue = {}) ⇒
    FDSpec.last_timeout ≥ FDImpl.T.clock)

  % Step Correspondence
  proof
    for internal send(m: M) ignore
    for internal receive(m: M) ignore
    for internal fail do fire internal fail od
    for output timeout do fire output timeout od
    for trajectory traj duration x do follow traj duration x od
```

```
Automaton initialized
1:   trajectory FDImpl.traj for 5.0 units
     trajectory FDSpec.traj for 5.0 units
2:   internal transition FDImpl.send(m1)
3:   trajectory FDImpl.traj for 2.0 units
     trajectory FDSpec.traj for 2.0 units
4:   internal transition FDImpl.receive(m1)
5:   trajectory FDImpl.traj for 3.0 units
     trajectory FDSpec.traj for 3.0 units
6:   internal transition FDImpl.send(m1)
7:   trajectory FDImpl.traj for 2.0 units
     trajectory FDSpec.traj for 2.0 units
8:   internal transition FDImpl.receive(m1)
9:   trajectory FDImpl.traj for 3.0 units
     trajectory FDSpec.traj for 3.0 units
10:  internal transition FDImpl.fail
     internal transition FDSpec.fail
11:  trajectory FDImpl.traj for 5.0 units
     trajectory FDSpec.traj for 5.0 units
12:  output transition FDImpl.timeout
     output transition FDSpec.timeout
13:  trajectory FDImpl.traj for Infinity units
     trajectory FDSpec.traj for Infinity units
No more steps
No errors
```

Figure 4-11: Failure Detection System Forward Simulation

Chapter 5

Discussion and Future Work

We discuss in this chapter some of the alternative directions the TIOA Simulator might have taken or could take in the future, as well as some suggestions on further improving the tool in the future.

5.1 Discussion

Restrictions to evolve predicates The TIOA Simulator currently restricts the form of evolve clauses to only simple differential equations of the form $d(x) = constant$. We discuss in Section 2.1 that this was not a problem for the examples we came across. An alternative solution, however, would be to allow the user to provide both the differential equation and its general solution, which would allow the simulator to calculate the values of the time-dependent variables. Further research is necessary, however, to make sure that the simulator will still be able to check the stopping conditions as well.

Scheduling input actions The TIOA Simulator allows firing input actions within an automaton's schedule. It might not be clear why an automaton's schedule is in control of its inputs. One way to think of this is to imagine as if an external `Environment` automaton that has the corresponding output actions exists, and a schedule in this `Environment` automaton is firing these actions. Allowing input actions therefore provides an easier way for a user to test their automata. A potential drawback of this decision is that the composite Simulator ignores some input actions and does not ignore others. In particular, it ignores the scheduled input actions that have corresponding output ones in other components of the system.

Another alternative could be to disallow firing input actions. This would result in simpler semantics for composite simulations, since input actions are not allowed altogether. On the other hand, this restriction will almost always require writing an `Environment` automaton specifically for the purpose of providing inputs to the system.

IOA Simulator features that are not supported Some of the features of the IOA Simulator are not part of the TIOA Simulator. We explain some of the reasons for each of them below:

- **Transition numbering (case identifiers)**. This feature allows simulating I/O automata with multiple transition definitions for the same action. For the range of examples we simulated this was not necessary, although it might become necessary in the future. This should be an easy extension to the TIOA Simulator.
- **The interface to the Daikon invariant detection system**. The IOA Simulator can optionally output information about the system as it is executed, to be fed into the Daikon invariant detection system. We have not considered how information about trajectory execution would be translated into Daikon input, so we did not expand this tool.
- **Smart fire**. The IOA Simulator allows for a "smart fire" statement in a schedule, which is simply the **fire** keyword for a special class of automata in which the actions have no parameters. The IOA Simulator then finds the set of enabled actions and fires one of them, chosen at random. The fact that most IOA (and TIOA) programs in the literature have parametrized actions lead to the decision to not include this feature, since it would not be used frequently.

5.2 Future Work

In this section we propose some of the important extensions to the TIOA Simulator that have not been yet implemented and would ideally be implemented in the future.

Nested composition Limitations in both the TIOA Checker and Simulator's implementation do not currently allow "nested compositions", i.e. simulation of a composite automaton whose components are also composite. This is certainly a limitation since it disallows multiple levels of composition, and a future extension to the Simulator should enable this feature.

Paired simulation of composite automata As illustrated in the failure detection example in Figure 4-10, paired simulation is currently restricted to primitive automata. It should be easy to extend the TIOA Simulator to support paired simulations where at least the implementation automaton is a composition of multiple automata.

Component referencing Declaring and using multiple components using an index is not currently implemented. Even though the Simulator supports creating multiple instances of an automaton, the user has to name each instance with a unique name and use that name within the schedules. A useful extension would be to use the parameters used for each instance as the reference names instead of the user-provided names. For example, suppose we create n instances of an automaton `P(i: Int)`, with the values $1...n$ for the formal parameter `i`. A particular component could then be referenced within the composite automaton's schedule by means of its index, e.g. `P(0)`. This would make schedules with large numbers of instances of the same automaton cleaner and shorter.

Graphical user interface A plug-in for the Eclipse Integrated Development Environment is under development. In the future, the prototype GUI of the TIOA Simulator (see Appendix A.3) can be merged into this plug-in.

In addition, one could assess different ways of displaying the results of the simulation. For example, an alternative way to displaying the execution trace is to display the actions of each automaton in a "time-line". The length of the time-line between two actions would be representative of the time that has passed between them. That portion of the line could also be labeled with the name of the trajectory that was followed. For composite automata, multiple parallel time-lines could be used, one for each component. Connected (shared) actions could be shown by drawing a link between the output action and the corresponding input ones. An example of how such a time-line could look like is shown in Figure 5-1

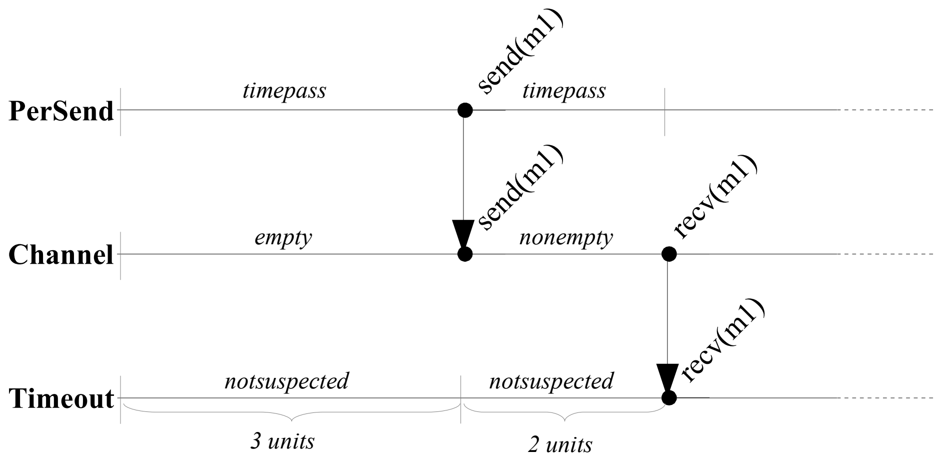


Figure 5-1: An example “time-line” view of an execution

Chapter 6

Conclusions

This Master's Thesis describes the design, implementation, and usage of the TIOA Simulator, a tool for testing and analyzing complex distributed systems. Based on the Timed Input/Output Automaton framework, the TIOA Simulator executes automata written in the TIOA Language. A set of restrictions and language extensions make execution of timed I/O automata possible with minimal additional effort from the user. The Simulator features simulation of both primitive and composite automata, as well as paired simulation of two related automata.

Simulation of TIOA is useful in the process of testing the proposed system over a specific set of executions. During the executions the Simulator is able to test proposed invariants and show that a proposed relation between the system's implementation and its specification is indeed a simulation relation. Testing the system, its invariants and its relation with the specification greatly increases the confidence about the system's correctness. Moreover, a formal verification of the system becomes easier since the proof strategies will follow naturally from the validated invariants, simulation relation and the step correspondence that drives the testing of the simulation relation.

The TIOA Simulator can therefore be used to find any problems of the system's implementation or specification, help the developer gain a better understanding of the system, facilitate verification and extract performance predictions such as message complexity. The combined use of the TIOA Simulator with the TIOA Checker and model checking or theorem proving tools provides a common formal framework for specifying, testing, and verifying complex distributed systems.

Appendix A

Configuration

A.1 Loading Runtime Classes

This section describes the configuration of the TIOA Simulator to enable loading of the runtime classes. Since TIOA allows users to specify their own data types and operators, the TIOA Simulator allows users to write implementations for these new data types and operators (in Java) and provides users a way to instruct the Simulator to locate and use them in runtime.

A.1.1 Command-line configuration file

Runtime configuration can occur by loading the options from a file. The command-line switch is `-config <configuration file>`. The file should contain the following options:

```
tioa.locationOfPackages:<list of paths to installation and to any other classes>
tioa.listOfPackages:<list of registration packages>
```

The paths should be absolute and the path separator should be the `/` character regardless of the platform. A sample configuration file is shown in Figure A-1. In this configuration the runtime classes are loaded from the TIOA Simulator's installation directory.

```
tioa.locationOfPackages:/opt/TIOASimulator/classes/
tioa.listOfPackages:tioa.registry.java
```

Figure A-1: Sample runtime configuration file

A.1.2 Default configuration file

A user can install a default configuration file that will be loaded every time the TIOA Simulator is run without the `-config` option.

The structure of the file is identical to that of a configuration file. The file should be in the user's home directory [1], under the name ".tioa"

A.2 Command-line Options

The TIOA Simulator expects the following command-line arguments:

```
[flags] <number of steps> [<automaton to simulate>] <tioa file>
```

A set of optional flags can be used to configure simulation. These flags are defined below:

- `[-config <string>]+` Loads the given configuration file(s) for runtime class location
- `[-dbg <string>]+` Enables debug information printing for a particular java class or package, e.g. `-dbg tioa.registry.ADT`
- `[-debug]` Enables global debug information printing
- `[-formals <string>]` Loads the formal parameter initial values from the given file
- `[-maxEnum <number>]` Sets the largest value to include when enumerating infinite types (currently only for enumerating Nat)
- `[-outputTraces]` Minimum level of verbosity: Output trajectories and only external transitions and no state variables.
- `[-outputTrans]` Medium level of verbosity: Output all trajectories and transitions and no state variables. Default level outputs all trajectories and transitions and only modified state variables.
- `[-outputState]` Maximum level of verbosity: Output all trajectories and transitions and all state variables.

[1] Usually `/home/username in Unix and C:\Documents and Settings\username in Windows`

- `[-randComp]` Composite Simulation: Choose which component to execute next randomly. Default is round-robin.

A.2.1 Formal parameters

The user can provide values for any formal parameters for the automaton to simulate during the runtime, using the `-formals` command-line option as shown above. The argument expected is the location of a file that contains the name of each parameter, its type and value, in the following syntax:

```
formals     ::=  '(' formal* ')'
formal      ::=  '(' formalName formalADT formalValue ')'
formalName  ::=  <identifier>
formalADT   ::=  <identifier>
formalValue ::=  <number>
```

The name of the formal parameter as used in the automaton's definition should be used for `formalName`, for example `u1`. The `formalADT` is the Java name of the runtime class that will be used for the formal parameter's data type. For built-in types, this is usually `tioa.runtime.adt.<type>Sort`, for example `tioa.runtime.adt.IntSort`. The formal's parameter value should be given in `formalValue`. Currently the simulator supports only literal numbers as values. An example of a formal parameter file is given below:

```
((a tioa.runtime.adt.IntSort 3)
 (b tioa.runtime.adt.RealSort 2))
```

A.3 Graphical User Interface

The command-line tool for the TIOA Simulator is useful for getting a trace of the execution, but a graphical user interface can make it easier for the developer to follow through the simulation, identify any problems and configure the simulation.

For this purpose we demonstrate a working prototype of a graphical user interface that acts as an integrated development environment for TIOA. It allows editing multiple files at the same time, provides syntax highlighting for TIOA and provides an interface to both the TIOA Syntax and Semantic Checker as well as the TIOA Simulator.

A.3.1 Checking specifications

A “Check” action, accessible from both the menu and the toolbar checks the specification for errors and displays the results in the “Problems” console. Checking the specification also results in displaying an outline of the specification in the left-hand side panel. The outline displays all the automata with their state variables, transitions and trajectories.

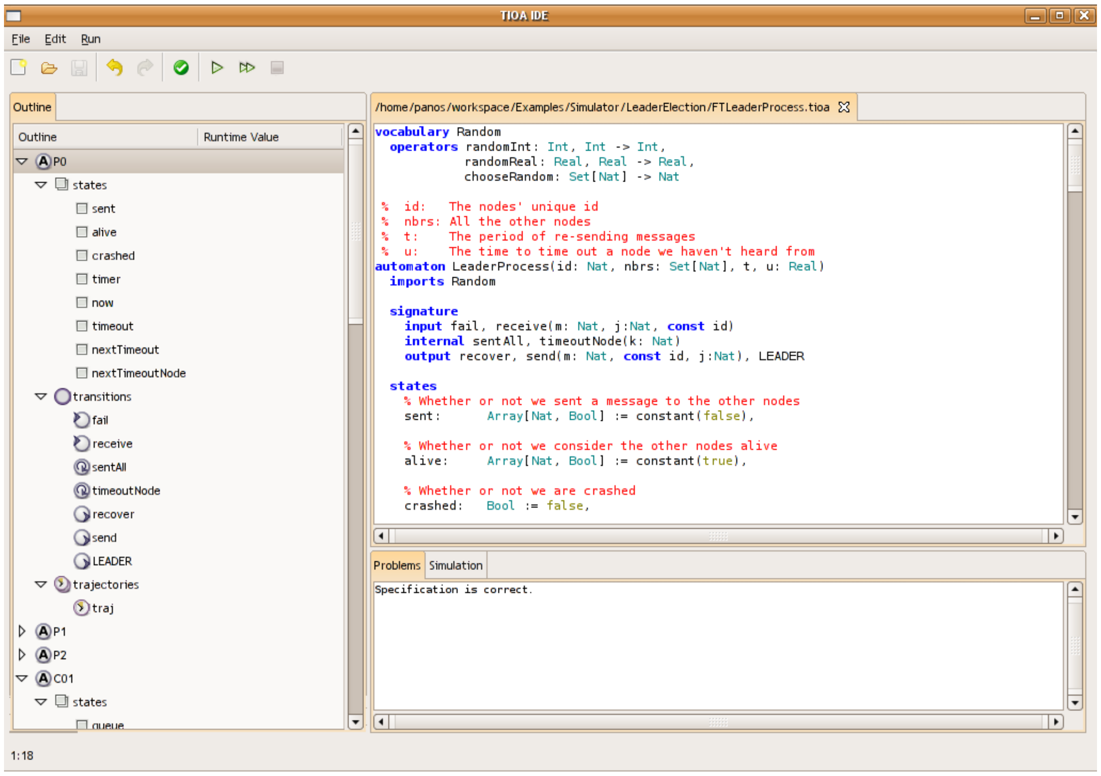
A.3.2 Simulation

For simulation we provide both a “step-by-step” action as well as a “run” action. The “step-by-step” action executes one step of the automaton and pauses. The user can continue with the next step by repeating the action. Step-by-step simulation can be interrupted at any point with the “Stop” action. The output of the simulator is shown in the “Simulation” console. At each step, the runtime values of the state variables are displayed in the “runtime value” column of the outline. The variables that were modified in the last step are displayed with bold typeface.

For quickly executing the simulation up to a maximum number of steps, the “Run” action can be used. The output of the simulation will appear in the “Simulation” console and the runtime values at the end of the execution will be visible in the outline.

Before the simulator begins executing, a simple dialog window will provide the user with the configuration options of the TIOA Simulator. In particular, it will provide a list of all the automata in the specification and ask the user which one to be used for the simulation, and also ask for other options such as the location of the formal parameter initial values file, the maximum number of steps to be taken, and the verbosity level of the output.

Figures A-2, A-3, A-4 provide some screen shots of the Simulator’s Graphical User Interface showing the outline view, the configuration dialog and the step-by-step features respectively.

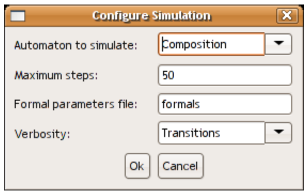


/home/panos/workspace/Examples/Simulator/LeaderElection/FTLeaderProcess.tioa

```
vocabulary Random
  operators randomInt: Int, Int -> Int,
            randomReal: Real, Real -> Real,
            chooseRandom: Set[Nat] -> Nat

 %  id:   The nodes' unique id
 %  nbrs: All the other nodes
 %  t:    The period of re-sending messages
 %  u:    The time to time out a node we haven't heard from
automaton LeaderProcess(id: Nat, nbrs: Set[Nat], t, u: Real)
  imports Random

  signature
    input fail, receive(m: Nat, j:Nat, const id)
    internal sentAll, timeoutNode(k: Nat)
    output recover, send(m: Nat, const id, j:Nat), LEADER

  states
    % Whether or not we sent a message to the other nodes
    sent:      Array[Nat, Bool] := constant(false),

    % Whether or not we consider the other nodes alive
    alive:     Array[Nat, Bool] := constant(true),

    % Whether or not we are crashed
    crashed:   Bool := false,
```

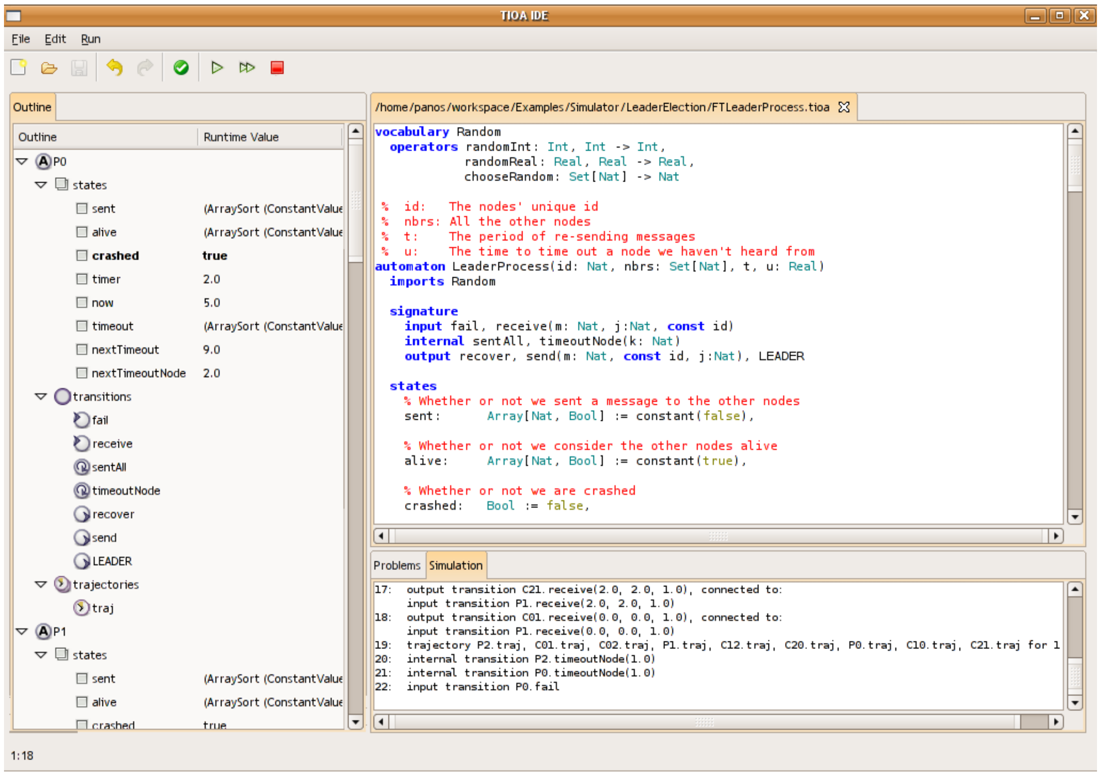
```
Specification is correct.
```

Figure A-2: The TIOA GUI with syntax highlighting and the outline view



Automaton to simulate:	Composition
Maximum steps:	50
Formal parameters file:	formals
Verbosity:	Transitions

Figure A-3: The TIOA GUI simulation configuration dialog


TIOA IDE
File
Edit
Run
Outline
/home/panos/workspace/Examples/Simulator/LeaderElection/FTLeaderProcess.tioa
Problems
Simulation
1:18

Outline	Runtime Value
P0	
states	
sent	(ArraySort (ConstantValue
alive	(ArraySort (ConstantValue
crashed	**true**
timer	2.0
now	5.0
timeout	(ArraySort (ConstantValue
nextTimeout	9.0
nextTimeoutNode	2.0
transitions	
fail	
receive	
sentAll	
timeoutNode	
recover	
send	
LEADER	
trajectories	
traj	
P1	
states	
sent	(ArraySort (ConstantValue
alive	(ArraySort (ConstantValue
crashed	true

```
vocabulary Random
  operators randomInt: Int, Int -> Int,
            randomReal: Real, Real -> Real,
            chooseRandom: Set[Nat] -> Nat

 %  id:   The nodes' unique id
 %  nbrs: All the other nodes
 %  t:    The period of re-sending messages
 %  u:    The time to time out a node we haven't heard from
automaton LeaderProcess(id: Nat, nbrs: Set[Nat], t, u: Real)
  imports Random

  signature
    input fail, receive(m: Nat, j:Nat, const id)
    internal sentAll, timeoutNode(k: Nat)
    output recover, send(m: Nat, const id, j:Nat), LEADER

  states
    % Whether or not we sent a message to the other nodes
    sent:      Array[Nat, Bool] := constant(false),

    % Whether or not we consider the other nodes alive
    alive:     Array[Nat, Bool] := constant(true),

    % Whether or not we are crashed
    crashed:   Bool := false,
```

```
17:  output transition C21.receive(2.0, 2.0, 1.0), connected to:
     input transition P1.receive(2.0, 2.0, 1.0)
18:  output transition C01.receive(0.0, 0.0, 1.0), connected to:
     input transition P1.receive(0.0, 0.0, 1.0)
19:  trajectory P2.traj, C01.traj, C02.traj, P1.traj, C12.traj, C20.traj, P0.traj, C10.traj, C21.traj for 1
20:  internal transition P2.timeoutNode(1.0)
21:  internal transition P0.timeoutNode(1.0)
22:  input transition P0.fail
```

Figure A-4: The TIOA GUI step-by-step simulation

Appendix B

TIOA Extensions

B.1 NDR Language

Automaton

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

Scheduling

```
NDRProgram      ::= NDRStatement;*
NDRStatement    ::= assignment | NDRConditional | NDRWhile |
                    NDRFire | NDRFollow | NDRYield
NDRConditional  ::= 'if' predicate 'then' NDRProgram
                    ('elseif' predicate 'then' NDRProgram)*
                    ('else' NDRProgram')? 'fi'
NDRWhile        ::= 'while' predicate 'do' NDRProgram 'od'
NDRFire         ::= 'fire' ('input' | 'output' | 'internal') actionName
                    actionActuals?
NDRFollow       ::= 'follow' trajectoryName 'duration' term
NDRYield        ::= 'yield' term
```

Choice

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

Simulation Relations

```
simulation      ::= ('forward' | 'backward') 'simulation' 'from'
                    automatonName 'to' automatonName ':'
                    predicate
                    Proof?
Proof           ::= 'proof' states? ('initially' (variable ':=' term);+)?
                     ProofEntry+
ProofEntry      ::= ProofTransEntry | ProofTrajEntry
ProofTransEntry ::= 'for' ('input' | 'output' | 'internal')
                    actionName actionFormals?
                    (('do' ProofProgram 'od') | 'ignore')
ProofTrajEntry  ::= 'for' 'trajectory' trajName 'duration'
                    durationVariable ':' durationType
ProofProgram    ::= ProofStatement;+
ProofStatement  ::= assignment | ProofConditional | ProofWhile |
                    ProofFire | ProofFollow
ProofConditional::= 'if' predicate 'then' ProofProgram
                    ('elseif' predicate 'then' ProofProgram)*
                    ('else' NDRProgram')? 'fi'
ProofWhile      ::= 'while' predicate 'do' ProofProgram 'od'
ProofFire       ::= 'fire' ('input' | 'output' | 'internal') actionName
                    actionActuals?  ('using' (term 'for' variable),+)?
ProofFollow     ::= 'follow' trajName 'duration' term
```

Bibliography

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

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

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

[4] Stephen Garland. TIOA user guide and reference manual. Manuscript, September 2005. URL `http://tioa.csail.mit.edu/public/Documentation/Guide.doc`.

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

[6] Dilsun Kaynar, Nancy Lynch, Sayan Mitra, and Stephen Garland. The TIOA language. Manuscript, October 2004. URL `http://theory.lcs.mit.edu/tds/papers/Kirli/TIOALanguage.pdf`.

[7] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed I/O automata. Synthesis Lectures on Computer Science, Morgan Claypool Publishers, 2005.

[8] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November 1988.

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

[10] Edward Solovey. Simulation of composite I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2003.

[11] Joshua A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2004.

[12] Joshua A. Tauber and Stephen J. Garland. Definition and expansion of composite automata in IOA. Technical Report MIT/LCS/TR-959, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 2004. URL `http://theory.lcs.mit.edu/tds/papers/Tauber/MIT-LCS-TR-959.pdf`.

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

[14] Toh Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 2003.

[15] Toh Ne Win, Michael Ernst, Stephen Garland, Dilsun Kirli, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 4:1–10, 2003.