# TIOA and UPPAAL

by

Christine Margaret Robson

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

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004



Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
20 May, 2004

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Stephen Garland
Principal Research Scientist
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Dilsun Kaynar
Postdoctoral Research Associate
Thesis Supervisor

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

**TIOA and UPPAAL**

by

Christine Margaret Robson

Submitted to the Department of Electrical Engineering and Computer Science
on 20 May, 2004, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

## Abstract

This thesis presents a tool that assists in the development of timed systems, giving users simulation and verification capacities to guarantee timing properties in computer programs. The IOA Toolkit, a toolset for developing distributed systems, is extended to permit time-dependant properties to be described using a subset of the Timed IOA (TIOA) language.

This thesis also includes a translator from TIOA into UPPAAL, another toolset. This translator allows the simulation of IOA and TIOA programs in UPPAAL's easy-to-use interface, and the checking of their properties with UPPAAL's model checker. This thesis lays the framework for an extension of the toolkit to utilize the full TIOA language, and to allow composed automata to be simulated. A scheme for translating composed automata into UPPAAL is also presented.

Thesis Supervisor: Stephen Garland
Title: Principal Research Scientist

Thesis Supervisor: Dilsun Kaynar
Title: Postdoctoral Research Associate

# Acknowledgments

Many people have contributed to this project, and it would not have been possible without them. My advisors, Dilsun Kaynar, Stephen Garland, and Nancy Lynch have provided guidance, support, and advice throughout this project. I would like to thank them for their help. Thanks also to Josh Tauber for his patient explanations on the inner workings of the IOA Toolkit.

My thanks also go to Karen, Mom, Dad, and Josh, for their invaluable support.

# Contents

# List of Figures

# Chapter 1

# Introduction

Time is of the essence. This statement is as true in today's computer industry as when it was first spoken. We rely on timely results from computer systems to manage many of the important systems in the backbone of our society. From managing power grids to printing letters, computer programs are entwined in much of our day-to-day lives, and we depend on their timely operation.

The systems we rely on are also becoming increasingly complicated. Complicated systems are naturally hard to trust; as a system gets larger, it is increasingly difficult to be convinced that it operates on time under all conditions. Exhaustive testing is often impossible; as in a game of chess, there are too many possible outcomes to examine every situation. When such complicated programs are used, however, guaranteeing timely responses is often important for safety, security, and performance.

The guarantee of timing properties in computer programs is the motivation for this thesis. This thesis project is part of a larger project, called IOA, in the Theory of Distributed Systems group at the MIT Computer Science and Artificial Intelligence Laboratories. The TDS group maintains and develops a toolset for developing distributed systems, the IOA Toolkit.

The IOA Toolkit is based on the formal Input/Output Automata model for program specification, and includes a formal specification language, simulator, verification tools, and interfaces to theorem provers. The IOA model for program specification was not designed to represent time-dependant systems. As such, the IOA Toolkit was not, until this thesis, usable for time-dependant systems. No timed systems could be verified or simulated, and no timing properties could be guaranteed.

In response to this limitation of the IOA modeling language and the IOA Toolkit, a new modeling language based on the formal Timed Input/Output Automaton framework has been introduced. This language is called TIOA and is based on IOA, with additional features to model time passage. A final version of a TIOA modeling language was decided during the course of this thesis.

The goal of this thesis was to extend the IOA Toolkit to assist with the development of timed systems. The IOA Toolkit was extended to parse a subset of the full TIOA language. For simulation and verification of such time-dependent systems, this thesis also includes an interface from the IOA Toolkit to UPPAAL, a model checking tool which models time and time constraints. This interface allows the translation of programs written in IOA and TIOA languages into UPPAAL. These programs must correspond to an UPPAAL-oriented restricted subset of the IOA and TIOA languages.

The UPPAAL translator allows IOA and TIOA programs to be simulated using UPPAAL's easy-to-use graphical interface. This translator ports primitive automata into UPPAAL. Primitive automata are distinct from composed automata, which are one or more automata with matching input and output actions, which execute together. As part of this thesis, the framework for simulating composed automata has been laid, and the methods described.

This thesis is a first step towards a complete extension of the Toolkit, which would allow the use of the full capacity of the IOA Toolkit's development tools in developing timed systems. The tools developed in this thesis project are particularly useful to users as a first-step check when designing time-dependant distributed systems, allowing them to catch conceptual errors early.

**Organization of the Thesis Document:** The background (Chapter 2) gives an overview of the three systems involved: Input/Output Automata (IOA), Timed IOA (TIOA), and UPPAAL. Following that, in Chapter 3, the interface between the IOA Toolkit and UPPAAL is described. Then, Chapter 4 describes the extension of the IOA Toolkit to allow TIOA, and the translation from TIOA to UPPAAL. Finally the issue of composition is discussed in Chapter 5, and the conclusion comprises Chapter 6.

# Chapter 2

# Background

To support the development of timed systems, this thesis presents an extension of the IOA Toolkit to allow time and timing properties to be represented, following TIOA syntax. This thesis also presents an interface between TIOA and UPPAAL. As a first step to providing a translator between TIOA and UPPAAL, a translator from IOA to UPPAAL was developed.

As a background to this thesis, the three systems involved, IOA, TIOA, and UPPAAL, are reviewed below.

## 2.1 Input/Output Automata (IOA)

As we have observed, the computer industry is awash with large and complicated systems, which require far longer to test then they did to develop. Many of these large systems employ distributed algorithms techniques- that is, the systems are made up of many similar (smaller) parts, which must interact with one another. There are many examples of such large distributed systems, a classic example being banks, with many similar branches and ATMs.

Given the size and complexity of these systems, it is difficult to guarantee execution properties. Complex systems are often impossible to test exhaustively, and exhaustive testing does not always reveal any execution properties of a program. This was one of the challenges which led to the IOA Project.

The IOA Project employs formal methods to rigorously analyze distributed systems with the help of computers [11, 36], including both a specification language, IOA [13], and a corresponding toolset [12].

The IOA language is a formal specification and programming language, which is based on the I/O Automaton model proposed by Nancy Lynch and Mark Tuttle [27, 28, 29]. It is particularly suited to distributed algorithms.

The IOA Toolkit is a toolset for developing distributed systems and includes verification and simulation tools. These tools allow users to simulate I/O Automata, and help the user to verify or disprove execution properties that the user wishes to test, by interfacing with theorem provers.

### 2.1.1 Language

The IOA Language models program executions as an alternating sequence of states and actions. Discrete steps in an execution are represented by transitions, which are tuples of the form $(s, a, s')$ where $a$ is an action, $s$ is the state before the action $a$ is preformed and $s'$ is the new state after the action a has been performed. Modeling programs in this way makes it easier to reason about their execution, and, more importantly, to demonstrate correctness. [13]

This "state-action-state" model is simple yet versatile. In order to specify a program, transitions can be described with preconditions and effects clauses. The precondition places limits on the possible initial states for an action to take place, whereas the effects clause specifies how the state will change once the action has taken place. A simple example of an IOA program is shown in Figure 2.1.1. This program runs an autonomous light switch, which turns on and off without human intervention.

Referring to Figure 2.1.1, there are three main sections in the specification of an automaton: *signature*, *states*, and *transitions*.

The *signature* lists the possible actions of the program by type, where actions can be

```
automaton Switch
  signature
    input  on
    output off
  states
    burning: Bool := false
  transitions
    input on
      eff burning := true
    output off
      pre burning = true
      eff burning := false
```

Figure 2.1.1: A simple IOA program, modeling a lamp switch

either *input*, *output*, or *internal*. In the light switch case, the program can accept an input command, to turn "on", and outputs notification that it has turned "off".

The *states* refer to the program variables, in which the state of the program is contained. The light switch only has one state variable: a boolean which indicates if the light is on or not.

The *transitions* describe the program's execution steps: each action described in the *transitions* must correspond to an action in the *signature* of the automaton. Transitions have optional precondition clauses, *pre*, which are predicates that need to be satisfied for the action to be enabled. If a transition has no *pre* clause, it may take place at any time. Input transitions are always enabled, thus they never have preconditions. For instance, in the light switch case, the input transition "on" may take place at any time.

Transitions also have optional effects clauses, *eff*, which modify the programs state by changing the value of state variables. Transitions with no *eff* clause make no change to the programs state, and as such, are rarely used.

This language has great flexibility; complicated programs can be expressed by creating multiple automata and "composing" them, so that the outputs of one automaton become the inputs of another. In this way, distributed algorithms can easily be modeled using multiple automata, one for each system component.

### 2.1.2 Toolkit

The IOA Toolkit provides a range of tools for developing distributed systems, with validation methods including not only simulation, but also proof checking [12].

The simulator has been developed by a number of members of the Theory of Distributed Systems group. Anna Chefter, as her Master's thesis [5] presented the initial design. Antonio Ramirez, proceeded with the implementation as his Master's thesis and built a paired simulation capability into the toolkit [35]. Edward Solovey extended the simulator to allow the simulation of composed automata, as part of his Master's thesis [37]. Dr. Stephen Garland has supervised and contributed to the simulators development.

Michael Tsai, in his Master's thesis, presented a means for IOA Programs to be compiled and simulated on a network of computers [39]. Josh Tauber, as part of his PhD thesis [38], is developing a composer that takes composite automata as input and produces and equivalent primitve automaton.

The toolkit is also connected to Daikon, a dynamic program analysis tool developed by Michael Ernst [9, 7, 8]. This feature of the Toolkit was begun by Laura Dean as her Master's thesis [6], and further developed by Toh Ne Win, as his master's thesis [30]. The Daikon interface provides a means of invariant detection, which is important for proof development in theorem provers, to guarantee program properties.

The IOA Toolkit also interfaces with theorem provers and model checkers. The interface to the LP Theorem Prover was written by Andrej Bogdanov's as his Master's Thesis [4]. The interface with the TLC model checker; the translation was written by Stanislav Funiak [10].

The validation methods available in the toolkit have been used to prove safety properties of distributed systems [31, 21, 26, 32, 33]. One such distributed system was a model for a bank. Other distributed algorithms that have been verified using the IOA Toolkit include leader election algorithms, tree spanning algorithms, and communication protocols.

## 2.2 Timed IOA (TIOA)

The IOA Project has until now been limited to time-independent algorithms; the IOA language as specified does not involve time. To address this issue, a new language, TIOA, has been designed [20, 19].

TIOA separates the notion of state change into two categories: discrete transitions model instantaneous changes to state, as in IOA, and trajectories model the evolution of state over time. The behavior of a program is then modeled as a sequence of actions and time-passages. Transitions take place instantaneously, so all time passage is modeled by trajectories.

The separate notions of actions and trajectories give TIOA the flexibility to model and analyze a broad range of real-time systems. This thesis only uses part of the full capabilities of TIOA. Specifically, only features of TIOA that can be harnessed for translation to UPPAAL are used.

### 2.2.1 Specification

TIOA is based on IOA; correspondingly, many of the major elements of the language are the same. The differences are the addition of trajectories to model time passage, and the addition of variables that measure time.

Trajectories are the basic model for time passage. A trajectory is specified by means of differential or algebraic equations that describe the evolution of continuous variables representing time [18]. It is possible to have continuous variables that change at varying rates. This means of specifying time passage is highly flexible, allowing the execution of non-real time systems with potentially very unusual time passages.

In this thesis, only real-time systems are considered. That is, systems in which time passage is linear, with rate one. If the continuous variable representing time is $t$, then the trajectory of the automaton must satisfy $d(t) = 1$, such that time is changing uniformly, at rate one. See [20] for a detailed explanation of the $d(.)$ notation.

Restrictions on time passage may be represented by stopping conditions or urgency predicates. Stopping conditions are expressed in the trajectory definition of an automaton, as

```
automaton Counter
        signature
                internal reset, count
        states
                x : int := 0,
                t : analog := 0
        transitions
                internal reset
                        eff x := 0;
                            t := 0
                internal count
                        pre x < 20
                        urgent when t = 5
                        eff x := x+1
        trajectories
                trajdef linear
                        evolve d(t) = 1
```

Figure 2.2.2: A TIOA program specifying a counter

they stop the trajectory from evolving. When a stopping condition is true, no time can pass until it becomes false.

Urgency predicates serve a similar function to stopping conditions, but are expressed differently, in the transitions of the automaton. When an urgency predicate in a transition definition becomes true, no time can pass until some transition takes place. This thesis deals only with urgency predicates, for reasons explained in Section 4.2.2. For a detailed discussion on the semantics of urgency predicates, please refer to [18] and [14].

### 2.2.2 Language

This thesis involves an UPPAAL-oriented, restricted version of the TIOA language, which is defined formally in Section 4.2. The language has an additional section to define trajectories, allows analog variables (clocks), and includes urgency predicates. An example of this TIOA code is shown in Figure 2.2.2. This toy example has 5 seconds to count as high as it can, up until the point when $x = 20$.

## 2.3 UPPAAL

UPPAAL is a tool environment for modeling, validating, and verifying real-time systems developed as a collaborative effort between Uppsala University, Sweden, and the University of Aarhus, Denmark. [3, 22, 34, 25]

The UPPAAL tool is typically used for verification of real time controllers and communication protocols. The key ability of UPPAAL is the ability to model time-passage during program execution. UPPAAL has the advantage of being able to verify time-dependant properties of a program.

UPPAAL is a useful target for interface with TIOA because it has proven itself as a model checking tool in the academic community. UPPAAL has been employed to verify protocols developed by universities, as well as several corporate groups, including Philips Audio [23, 2], Lego Mind Storms Systems [17], and Bang Olufsen[16, 15].

### 2.3.1 Simulator

UPPAAL is a popular model checking system, not only because of its verification and simulation capabilities, but also because of its well designed user-interface (see Figure 2.3.3). Programs are represented in a graphical interface that models a state machine, where individual states of the program are represented as nodes, called locations. Changes in state are represented by arrows called transitions, which point from one location to another. These transitions have preconditions and effects clauses similar to IOA.

UPPAAL also allows variables, such as clocks, integers, and booleans, which are not represented graphically. The transitions may refer to these variables in preconditions and effects clauses. When simulated, these variables are updated with each step of the program, in a side window.

It is important to make a distinction between the IOA notion of state and the UPPAAL notion of state. When we refer to state in the context of IOA, we are referring to a snapshot of the program's state, including the values of all state variables. In UPPAAL, state refers to a single location in the graphical interface. The values of the variables are independent

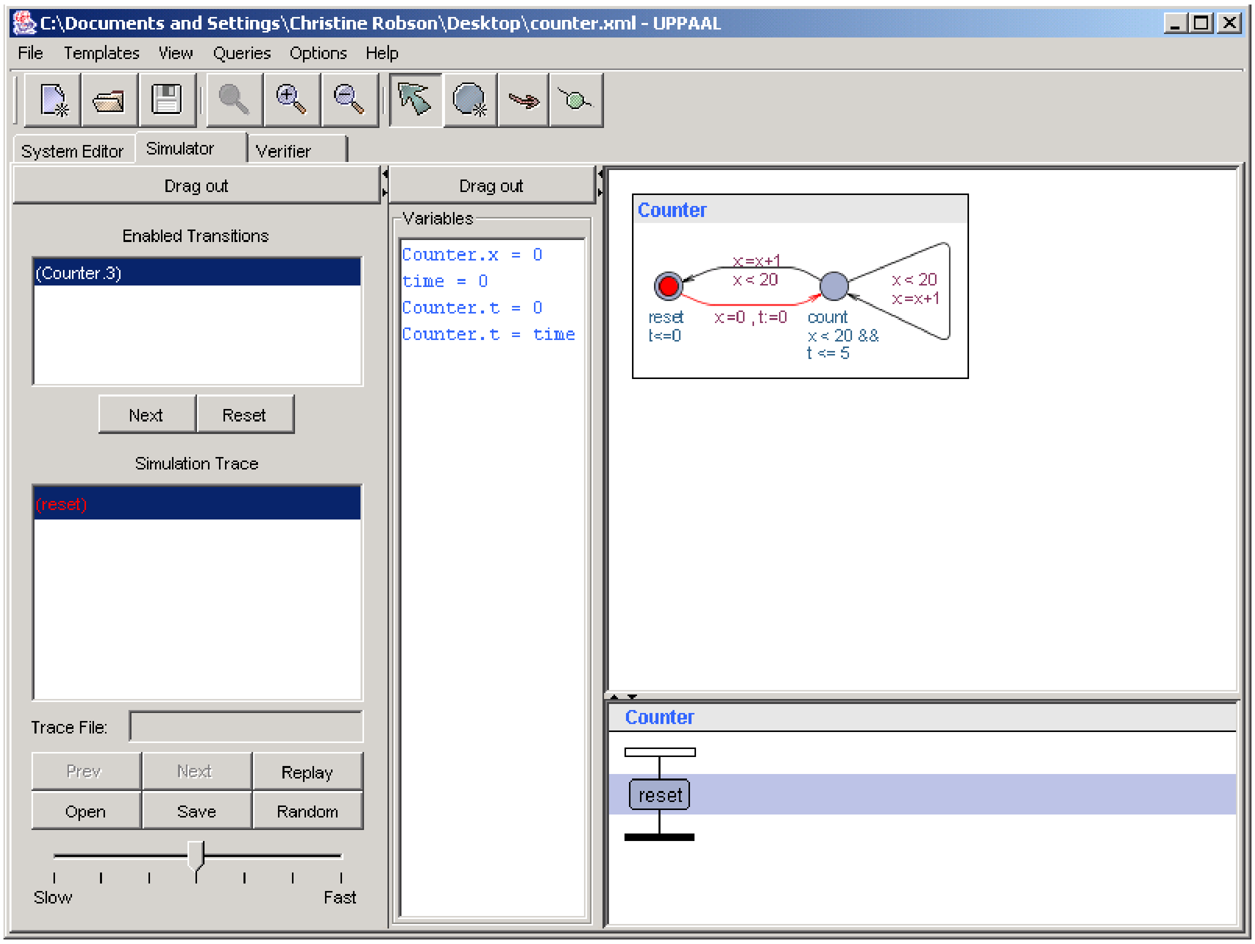


Figure 2.3.3: Screenshot of the UPPAAL Simulator

of these states. Thus an UPPAAL state may correspond to multiple IOA states, if there are state variables which may take on different values [22].

One very useful graphical feature of UPPAAL is the execution trace: when simulating execution of the program, the execution trace is shown graphically as a series of connected boxes, representing the state the program has transitioned through to reach its current point. This is shown in the UPPAAL screenshot in Figure 2.3.3.

### 2.3.2 Program Specification

All programs specified in UPPAAL conform to a basic structure. A "Project" contains all the information about a single program. This project may contain one or more "Processes," which represent individual pieces of the program. Processes are described by a set of "Locations", with "Transitions" connecting them. Variables may be defined globally, within the project, or locally, within each process.

To illustrate the structure of UPPAAL-specified programs, an inheritance diagram is available as Figure 2.3.4. This diagram shows both how individual parts of the program are represented in the graphical user interface, and also how the parts are linked to one another.

### 2.3.3 File Formats

UPPAAL employs two file formats to save UPPAAL programs: XTA and XML. These are both plaintext file formats, which contain all the parameters of the program.

As of UPPAAL version 2K, the default file format is XML [34, 25]. The XML file format contains embedded graphical information; the pictorial layout of the program is stored in the same file as the program specification.

The older file format for UPPAAL is XTA, which breaks down all the parameters of the program by type. XTA does not contain any graphical information; an accompanying file in CGI format specifies the location of each state and transition, as well as all labels in the graphical interface.

In the absence of a CGI file, it is still possible to load an XTA file. UPPAAL contains an auto-plot function, which plots the XTA-specified program according to a grid, and can then create the required CGI file. The resulting programs are easily readable.

To interface the IOA Toolkit with UPPAAL, the XTA file format for UPPAAL programs was chosen as a target. This decision was made to avoid the difficulty of providing graphical layout information for the UPPAAL programs. Since UPPAAL already contains an auto-plot feature, there is nothing to be gained by re-implementing that system.

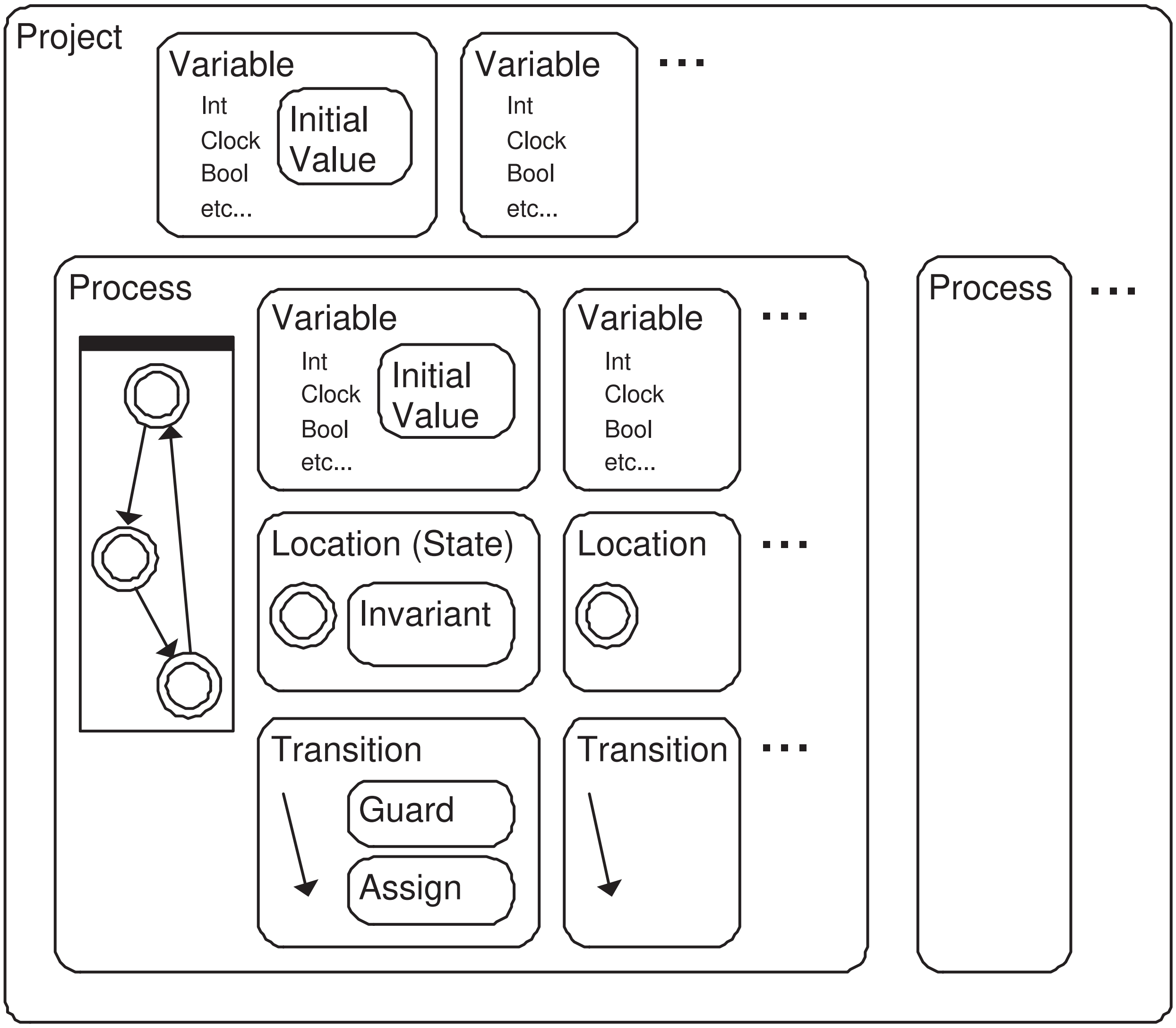


Figure 2.3.4: UPPAAL-Specified Program Structure

```
clock time;
process Counter{
  clock t;
  int x := 0;
  state reset{t≤0}, count{x < 20 && t ≤ 5};
  init reset;
  trans
    count → count{guard x < 20; assign x:=x+1; },
    count → reset{guard x < 20; assign x:=x+1; },
    reset → count{assign x:=0 , t:=0; };
}
system Counter;
```

Figure 2.3.5: Example UPPAAL Code in XTA Format for a Counter

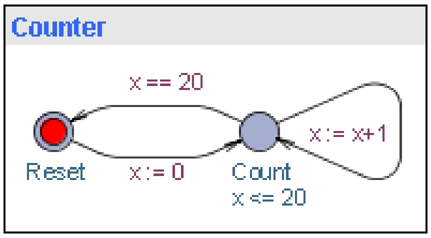


Figure 2.3.6: Resulting UPPAAL Program from Code in Figure 2.3.5

### 2.3.4 XTA Specification

XTA is not a file format meant to be read by humans. However, it provides an easy way to import IOA programs into UPPAAL. Figure 2.3.5 contains a small example which demonstrates the format, and Figure 2.3.6 shows the resulting graphical rendition of the program. The program in these figures is the counter from Figure 2.2.2, which has 5 seconds to count as high as it can, up until the point when $x = 20$.

Taking a closer look at the XTA specifications, we see that the file format is broken down into several sections. First the global variables are defined:

```
clock time;
...
```

Then the processes, that is, single programs that may interact with one another, are defined. This example has only one process. The processes contain local variables, as well as states and transitions:

```
...
process Counter{
  clock t;
  int x := 0;
  state reset{t ≤ 0}, count{x < 20 && t ≤ 5};
  init reset;
  trans
    count → count{guard x < 20; assign x:=x+1; },
    count → reset{guard x < 20; assign x:=x+1; },
    reset → count{assign x:=0 , t:=0; };
}
system Counter;
```

The "init" flag indicates the initial or start state of the program. All programs have an initial state. In this example *reset* is the initial state. Transitions are specified directionally between two states, with "guard" and "assign" clauses corresponding to the preconditions and effects of the action, respectively.

Invariants on states are specified in brackets after the state is declared. Invariants conform to a few specifications when they involve time; these requirements are discussed in Section 4.2. In this example, the invariant of *count* is $x < 20 \&\& t <= 5$, indicating that counting can only take place if $x < 20$ and $t <= 5$.

### 2.3.5 Model Checking

UPPAAL includes a verification tool which allows model checking of UPPAAL programs. This tool checks all possible execution paths of the program, and reports if specified invariants hold true. To use this verification tool, users must enter properties to check in a specified format, as outlined in Figure 2.3.7.

From Figure 2.3.7, $p$ and $q$ are typically UPPAAL states, declared by *ProcessName.LocationName*, or conditions on variables, such as $t < 10$ [22].

Referring to Figure 2.3.5, the counter which has 5 seconds to count as high as it can, up until the point when $x = 20$, we can confirm the following invariants:

$A[\,]\ x < 20:$ the program never counts higher then 20
$A[\,]\ t <= 5:$ the program never counts longer then 5 seconds

| | |
|---|---|
| $E <> \ p$ | there exists a path where p eventually holds |
| $A[\,]\ p$ | for all paths p always holds |
| $E[\,]\ p$ | there exists a path where p always holds |
| $A <> \ p$ | for all paths p will eventually hold |
| $p\ \ --> \ q$ | whenever p holds, q will eventually hold. |
| $p$ imply $q$ | whenever p holds, q holds. |

Figure 2.3.7: Syntax for Specifying Properties in UPPAAL Verification Tool

# Chapter 3

# IOA to UPPAAL Translation

## 3.1 Overview

As a first step towards extending the IOA toolkit to accommodate TIOA, and to creating a portal between TIOA and UPPAAL, this project included a translation from IOA to UPPAAL. Since IOA does not involve time, this chapter does not involve the translation of any timing constraints or properties.

The translation of IOA to UPPAAL began with the selection of some IOA programs to translate into UPPAAL. Three programs were selected: Squareroot; Integer Division; and Channel, a message channel. These programs were initially hand translated into UPPAAL.

The translation of these three programs was used to generate a translation scheme for IOA to UPPAAL. This translation scheme was used to create the translator from IOA to XTA. This ioa2xta translator was used to translate the original three programs, as well as the Peterson exclusion algorithm for two processes, for assessment of the translator.

## 3.2 Requirements on IOA

The ioa2xta translator is implemented for a subset of the complete IOA language; some restrictions were necessary to accommodate UPPAAL, and to limit the scope of this initial project. The bread-and-butter aspects of the language are preserved; however some language constructs whose translation would not be paractical in UPPAAL are not permitted by the translator. For example, conditional statements, choice statements, and for loops.

### 3.2.1 Variable Restrictions

When compared to IOA, UPPAAL has a limited vocabulary of variables. The only variables that both IOA and UPPAAL allow are integers, booleans (represented as integers 1 and 0), and arrays of integers or booleans. Enumerated variables are allowed, since they are translated into integers, with possible values between 1 and $n$, for an enumeration of $n$ elements.

These variables, and no others, are permitted in the restricted IOA language which is used by the translator.

### 3.2.2 Operator Restrictions

Logical and arithmetic operators are only permitted if they exist in both UPPAAL and IOA. These operators are outlined in Figure 3.2.1.

### 3.2.3 Program Restrictions

The ioa2xta translator is used for primitive, closed automata only. UPPAAL allows no inputs to programs during simulation. Thus, all IOA automata must be closed. This means automata can not have input actions. If an automaton takes on an input during execution, the automaton must be re-written so that the input is hard-coded. Moreover, transition definitions are not allowed to have parameters.

Automatic translation of composite automata is outside the scope of this project, however a preliminary scheme for translating composite automata is presented in Section 5.

<table>
<tr><th>IOA</th><th>XTA</th><th>Meaning</th></tr>
<tr><td>()</td><td>()</td><td>Parenthesis</td></tr>
<tr><td>[]</td><td>[]</td><td>Array lookup</td></tr>
<tr><td>~</td><td>!, not</td><td>Logical negation</td></tr>
<tr><td>−</td><td>−</td><td>Integer subtraction</td></tr>
<tr><td>+</td><td>+</td><td>Integer addition</td></tr>
<tr><td>*</td><td>*</td><td>Integer multiplication</td></tr>
<tr><td>/</td><td>/</td><td>Integer division</td></tr>
<tr><td><</td><td><</td><td>Less than</td></tr>
<tr><td><=</td><td><=</td><td>Less than or equal to</td></tr>
<tr><td>=</td><td>==</td><td>Equality operator</td></tr>
<tr><td>~=</td><td>!=</td><td>Inequality operator</td></tr>
<tr><td>>=</td><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td>></td><td>></td><td>Greater than</td></tr>
<tr><td>/\</td><td>&&, and</td><td>Logical and</td></tr>
<tr><td>\/</td><td>||, or</td><td>Logical or</td></tr>
</table>

Figure 3.2.1: Permissable Operations for IOA Files Inputed to the ioa2xta Translator

## 3.3 Translation Scheme

All variables in IOA become variables in UPPAAL.

Actions in IOA become transitions in UPPAAL. The IOA precondition maps to the guard clause and the IOA effect maps to the assign clause.

Each program has a single Location. All transitions have this location as both their source and target. This location is called *default*.

An alternative method of translation which was considered was using multiple locations, when automata have enumerated variables. In this scheme, each location would represent one value of an enumerated variable. This scheme provided accurate translations with easy-to-read graphical representations in UPPAAL, for automata which had enumerated variables.

This alternate translation scheme was abandoned during research into the TIOA to UPPAAL translator. As explained later, in Chapter 4, in order to represent the urgency predicates of TIOA in UPPAAL, it is necessary to use location invariants. The modeling of multiple locations for enumerated variables would directly interfere with the representation of urgency predicates. In order to use the ioa2xta translator as a basis for the tioa2xta translator, the notion of multiple locations for a single I/O automata was abandoned.

Thus the final translation scheme chosen for IOA to UPPAAL employs a *default* location. Pseudo-code for this final translation scheme is presented in Figure 3.3.2.

## 3.4 Implementation

The IOA Toolkit parses an IOA file describing an automaton, and creates a Java tree with an automaton object at the root. The ioa2xta program makes use of this feature, by using the IOA Toolkit parser to generate the Java tree.

### 3.4.1 UPPAAL package

To capture the information contained in an XTA file, a package was added to the IOA Toolkit. This package, uppaal, models the objects in UPPAAL (i.e. states, transitions,

```
AutomatonStrip(Automaton a){
  Project proj = new Project();
  for all StateVariable s in a {
    Variable v = new Variable(s);
    proj.add(v);
  }
  Process p = new Process();
  Location l = new Location();
  p.addLocation(l);
  for all Transition t in a {
    UppaalTransition ut = new UppaalTransition(t);
    ut.setGuard(t.precondition);
    ut.setAssign(t.effects);
    p.addTransition(l, ut, l);
  }
  proj.add(p);
}
```

Figure 3.3.2: Pseudocode for translation from IOA to XTA

processes, e.t.c.), in the proper hierarchy.

The structure of this package is diagrammed in Figure 3.4.3. Every UPPAAL program has a project at its root. Projects can contain multiple processes; however, in the ioa2xta translation, an automaton is modeled as one process. The model of one-process for one automaton is further described in Section 5, Composition with UPPAAL. Similarly, each process may contain multiple locations (UPPAAL states or UStates); however iao2xta, generates only a single UState. Locations may have invariants, a feature which is not used by ioa2xta, but is used by the TIOA to UPPAAL translator.

Processes also have local variables and transitions. Transitions in turn have guards and assigns, which correspond to preconditions and effects of an action.

### 3.4.2 Translation

As described above, the ioa2xta translator uses the existing features of the IOA Toolkit to parse an IOA file into a Java tree, with an Automaton at its root. The translator walks through this tree, creating a corresponding tree in the uppaal package. This translator called

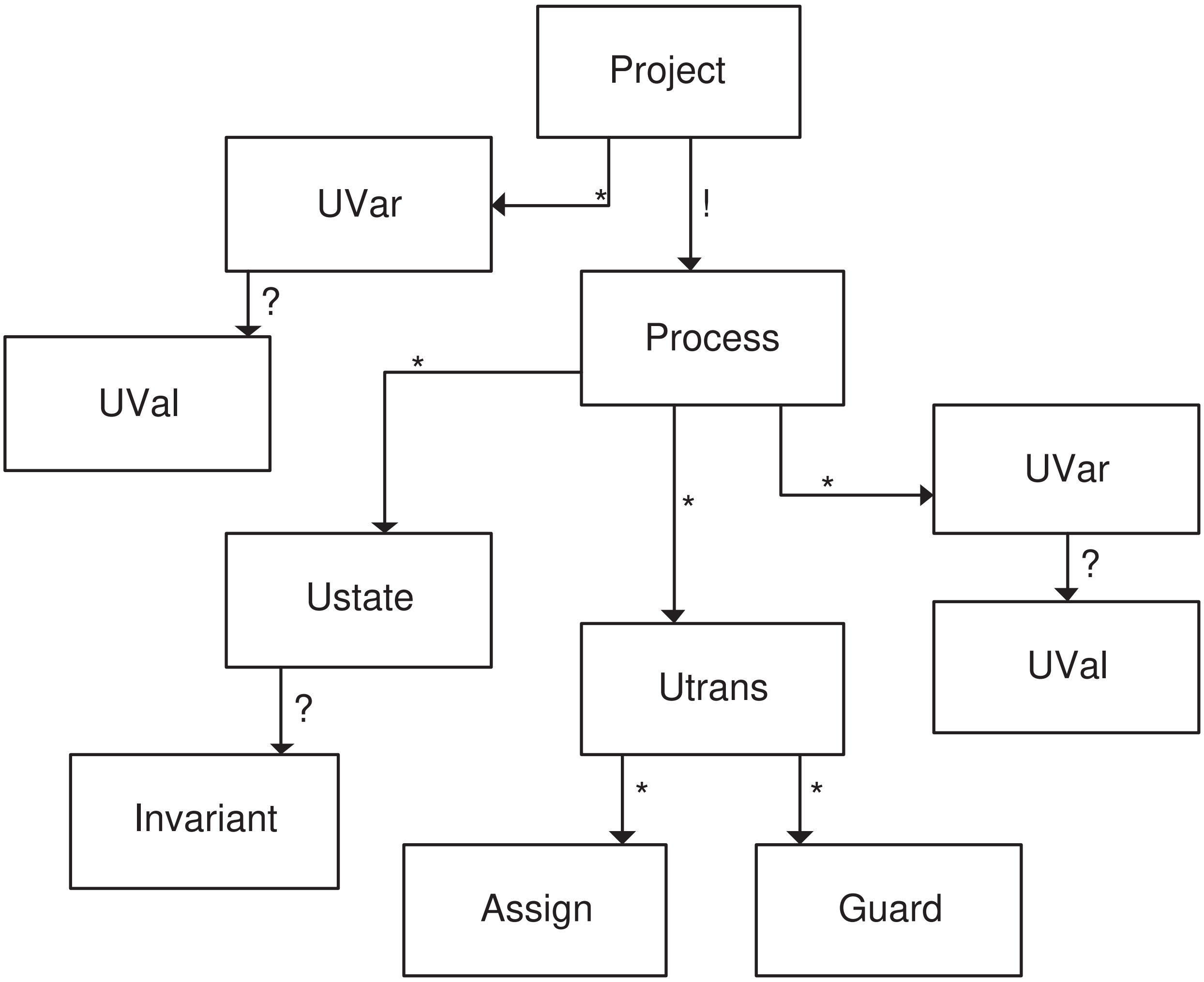


Figure 3.4.3: Classes and dependancies in the uppaal package of the IOA Toolkit

AutomatonStrip. Pseudocode for AutomatonStrip is presented in Figure 3.3.2.

Each class in the uppaal package contains a function called printXTA(). When printXTA() is called on a Project, it recursively calls printXTA() on all other objects in its tree, and then returns a string that contains the specification of the project in XTA format.

This printXTA() function is used to output an XTA file that can be read by UPPAAL.

## 3.5 Testing the Translation

The Peterson exclusion algorithm for two processes, being a sufficiently complex algorithm and a good example of a distributed system, was chosen as a test case for translation. The translation is shown in Section 3.6.4. The IOA code for the Peterson program is shown in Figure 3.6.13 on page 45.

The ioa2xta translator outputted an XTA file, shown in Figure 3.6.14. The resulting program was run in the UPPAAL simulator and verifier. As described in Section 3.6.4, the invariants of the Peterson exclusion algorithm were preserved.

```
automaton SquareRoot
  signature
    internal compute, halt
  states
      x: Nat := 225,           z: Nat := 0,
      u: Nat := 1,             done : Bool := false
      w: Nat := 1,
  transitions
    internal compute
      pre ¬done ∧ w ≤ x
      eff z := z+1 ; u := u+2 ; w := w+u
    internal halt
      pre ¬done ∧ w > x
      eff done := true
```

Figure 3.6.4: IOA Code for Squareroot

## 3.6 Examples

### 3.6.1 Squareroot

This program operates by finding the biggest square number bigger then $x$, and outputting that number in $w$, and the squareroot of $w$ in $z$. The variable $u$ is used for calculations. Because UPPAAL programs cannot take any inputs during execution, $x$ has been preset to 225. The IOA code for Square Root is shown in Figure 3.6.4, and the resulting XTA output from the translation is shown in Figure 3.6.5.

This example is mathematical and does not have particularly interesting properties to verify. It was chosen to model translation involving real numbers. Some properties of the translated program were verified using the UPPAAL verification tools: the value of $w$ is bounded by 0 and $x^2$ at all times and when *done* is true, $w > x$, and $w = z^2$. These properties are shown in Figure 3.6.6.

### 3.6.2 Integer Division

This program calculates $x/y = z$, with remainder $w$. Since UPPAAL programs allow no inputs during simulation, $x$ and $y$ must be preset. In this example, the program calculates

```
int x := 225;
int u := 1;
int w := 1;
int z := 0;
bool done := false;
process SquareRoot{
  state default;
    init default;
  trans
    default → default{guard done == false, w ≤x;
      assign z := z+1, u := u+2, w := w+u;},
    default → default{guard done == false, w >x;
      assign done := true;};
}
system SquareRoot;
```

Figure 3.6.5: XTA Code for Squareroot

$A[\,]\ w\ >=\ 0$
$A[\,]\ w\ >=\ x * x$
$A[\,]\ \mathit{done}\ =\ \mathit{true}$ imply $w\ >\ x$
$A[\,]\ \mathit{done}\ =\ \mathit{true}$ imply $w\ =\ z * z$

Figure 3.6.6: Invariants of Squareroot

```
automaton IntegerDivision
  signature
    internal initialize, compute1, compute2, halt
  states
     x : Int := 313,         z : Int,        initial : Bool := true,
     y : Int := 12,          w : Int,        done : Bool := false
                             t : Int,
  transitions
    internal initialize
      pre initial
      eff t:= x; z:= 0; w:= 0; initial := false
    internal compute1
      pre ¬initial ∧ ¬done ∧ t>0 ∧ w+1 = y
      eff z:=z+1 ; w := 0 ; t := t-1
    internal compute2
      pre ¬initial ∧ ¬done ∧ t>0 ∧ w+1 ≠ y
      eff else w := w+1 ; t := t-1
    internal halt
      pre ¬init ∧ ¬done ∧ t ≤ 0
      eff done := true
```

Figure 3.6.7: IOA Code for Integer Division

313/12. The IOA code for Division is shown in Figure 3.6.7, and the resulting XTA output from the translation is shown in Figure 3.6.8.

Like squareroot, this example is mathematical. The UPPAAL verification tools were used to verify the properties that $x > w$, if $x > y$ then $z > 1$, and when *done* is true, $z*y+w = x$. These properties are shown in Figure 3.6.9.

```
int x := 313;
int y := 12;
int z;
int w;
int t;
bool initial := true;
bool done := false;
process IntegerDivision{
  state default;
    init default;
  trans
    default → default{guard initial==true;
      assign t:=x, z:=0, w:=0, initial:=false;},
    default → default{guard initial==false, done==false, t>0, w+1==y;
      assign t := t-1, z := z+1, w:=0;},
    default → default{guard initial==false, done==false, t>0, w+1!=y;
      assign t := t-1, w:= w+1;},
    default → default{guard initial==false, done==false, t≤0;
      assign done := true;};
}
system IntegerDivision;
```

Figure 3.6.8: XTA Code for Integer Division

$A[\,]\; x \;>=\; w$
$A[\,]\; x \;>\; y$ imply $z \;>\; 1$
$A[\,]\; done \;=\; true$ imply $z * y + w \;=\; x$

Figure 3.6.9: Invariants of Division

```
automaton Channel
  signature
    internal send
    internal receive
  states
    queued: integer := 12,
    sent: integer := 0,
    intransit: boolean := false,
    received: integer := 0
  transitions
    internal send
      pre queued > 0 ∧ ¬intransit
      eff queued := queued - 1;
          sent := sent + 1;
          intransit := true
    internal receive
      pre intransit
      eff received := received + 1;
          intransit := false
```

Figure 3.6.10: IOA Code for a Channel

### 3.6.3 Channel

This channel program has a fixed queue of messages, since UPPAAL programs cannot take any inputs during execution. This channel sends one message at a time and has 12 messages to send. The IOA code for Channel is shown in Figure 3.6.10, and the resulting XTA output from the translation is shown in Figure 3.6.11.

The channel example is a more useful example then Squareroot or Division, since channels are common components of distributed systems. As such, channels are often modeled in the IOA Toolkit. This channel was simulated in UPPAAL, and the properties in Figure 3.6.12 were verified using the UPPAAL verification tools.

```
int queued := 12;
int sent := 0;
bool intransit := 0;
int received:= 0;
process Channel{
  state default;
    init default;
  trans
    default → default{
      guard queued > 0, intransit == false;
      assign queued := queued - 1, sent:= sent + 1, intransit := true; };
    default → default{
      guard intransit == true;
      assign received := received + 1, intransit := false; },
}
system Channel;
```

Figure 3.6.11: XTA Code for a Channel

$A[\,]\ queued\ +\ sent = 12$
$A[\,]\ intransit\ =\ false$ imply $sent\ =\ received$
$A[\,]\ intransit\ =\ false$ imply $queued\ +\ received\ =\ 12$
$A[\,]\ intransit\ =\ true$ imply $sent\ =\ received\ -\ 1$
$A[\,]\ intransit\ =\ true$ imply $queued\ +\ received\ =\ 11$

Figure 3.6.12: Invariants of Channel

### 3.6.4 Peterson Exclusion

The Peterson exclusion algorithm is a classic example of a distributed algorithm. In this example we show it for two processes. The IOA code for the Peterson exclusion algorithm with two processes is shown in Figure 3.6.13, and the resulting XTA output from the translation is shown in Figure 3.6.14.

Unlike the previous examples, the Peterson algorithm was not initially hand translated before it was automatically translated. Instead it was reserved as a test program to see if the translation scheme developed by hand-translating the previous three programs was successful.

The Peterson exclusion algorithm was simulated in UPPAAL, and the properties in Figure 3.6.15 were verified using the UPPAAL verification tools. The verification of these properties serves as a successful test of the translator.

```
type PCType = enumeration of waiting0, trying0,
              trying1, trying2, critical0, critical1

automaton Peterson
signature
  output trying_p0, trying_p1
  internal setFlag_p0, setFlag_p1, setTurn_p0, setTurn_p1,
    checkFlag_p0, check_Flag_p1, checkTurn_p0, checkTurn_p1
  output critical_p0, critical_p1, release_p0, release_p1
states
  p0 : PCType := waiting0,
  p1 : PCType := waiting0,
  flag0 : Bool := false,
  flag1 : Bool := false,
  turn : Int := 1
transitions

   output trying_p0
   pre
     p0 = waiting0
   eff
     p0 := trying0

   output trying_p1
   pre
     p1 = waiting0
   eff
     p1 := trying0

   internal setFlag_p0
   pre
     p0 = trying0
   eff
     p0 := trying1;
     flag0 := true

   internal setFlag_p1
   pre
     p1 = trying0
   eff
     p1 := trying1;
     flag1 := true

   internal setTurn_p0
   pre
     p0 = trying1
   eff
     p0 := trying2;
     turn := 0

   internal setTurn_p1
   pre
     p1 = trying1
   eff
     p1 := trying2;
     turn := 1

   internal checkTurn_p0
   pre
     p0 = trying2
     turn = 1
   eff
       p0 := critical0

   internal checkTurn_p1
   pre
     p1 = trying2
     turn = 0
   eff
     p1 := critical0

   internal checkFlag_p0
   pre
     p0 = trying2
     flag1 = false
   eff
     p0 := critical0

   internal checkFlag_p1
   pre
     p1 = trying2
     flag0 = false
   eff
     p1 := critical0

   output critical_p0
   pre
     p0 = critical0
   eff
     p0 := critical1;

   output critical_p1
   pre
     p1 = critical0
   eff
     p1 := critical1;
```

Figure 3.6.13: IOA Code for Peterson 2P

```
int flag0;
int p0;
int turn;
int flag1;
int p1;
process defaultProcess{
state defaultState;
init defaultState;
trans defaultState → defaultState{guard p1=1; assign p1:=2, flag1:=1; },
defaultState → defaultState{guard p0=4; assign p0:=5; },
defaultState → defaultState{guard turn=0, p0=3; assign p0:=4; },
defaultState → defaultState{guard flag1=0, p0=3; assign p0:=4; },
defaultState → defaultState{guard p0=0; assign p0:=1; },
defaultState → defaultState{guard p0=2; assign p0:=3, turn:=0; },
defaultState → defaultState{guard p1=5; assign p1:=0, flag1:=0; },
defaultState → defaultState{guard p1=0; assign p1:=1; },
defaultState → defaultState{guard p1=3; assign p1:=3, turn:=1; },
defaultState → defaultState{guard turn=0, p1=3; assign p1:=4; },
defaultState → defaultState{guard p1=4; assign p1:=5; },
defaultState → defaultState{guard p0=5; assign p0:=0, flag0:=0; },
defaultState → defaultState{guard p0=1; assign p0:=2, flag0:=1; },
defaultState → defaultState{guard flag0=0, p1=3; assign p1:=4; };
}
system defaultProcess;
```

Figure 3.6.14: XTA Code for Peterson 2P

$E <> \ turn = 0$ : for some execution path, process 0 gets a turn
$E <> \ turn = 1$ : for some execution path, process 1 gets a turn
$A[\,]\ p0 \ = \ 1 \ --> \ turn = 0$ : if process 0 is trying, it will eventually get a turn
$A[\,]\ p1 \ = \ 1 \ --> \ turn = 1$ : if process 1 is trying, it will eventually get a turn
$A[\,]\ p0 \ = \ 3$ impy $turn = 0$ : if process 0 is critical, it gets a turn
$A[\,]\ p1 \ = \ 3$ impy $turn = 1$ : if process 1 is critical, it gets a turn

Figure 3.6.15: Properties of the Peterson 2P Program

# Chapter 4

# Timed IOA (TIOA)

## 4.1 Overview

This chapter describes the implementation of the project for time dependant systems: namely, extending the IOA toolkit to accommodate TIOA, and creating a portal between TIOA and UPPAAL.

The extension of the toolkit to allow TIOA is described in Section 4.3.

The translator from TIOA to XTA, tioa2xta, is based on the IOA to XTA translator, described in Section 3.3. Similarly to the development of ioa2xta, the development of tioa2xta began with the selection of some TIOA programs to translate into UPPAAL. The translation of these programs was used to generate a translation scheme.

Throughout this chapter we will refer to an example program, Train. The Train program models a train approaching a crossing. The light is signaled, then the gate is lowered for the train to cross (see Figure 4.1.1). Each of these actions takes place with a certain urgency. The TIOA code for this train is presented in Figure 4.1.2 and the XTA code in Figure 4.1.4.

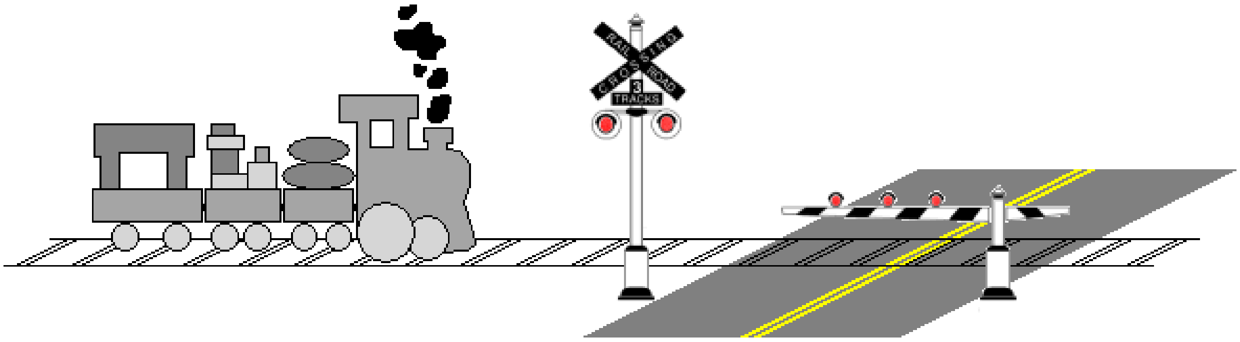

Figure 4.1.1: Train Example

```
type ControlType = enumeration start, light, gate

automaton Train
        signature
                internal coming, approaching, passing
        states
                control : ControlType := start,
                t : analog := 0
        transitions
                internal coming
                        pre t > 2 ∧ control = start
                        urgent when t = 5
                        eff control := light; t := 0
                internal approaching
                        pre t > 5 ∧ control = light
                        urgent when t = 10
                        eff control := gate; t := 0
                internal passing
                        pre t = 2 ∧ control = gate
                        urgent when true
                        eff control := start; t := 0
        trajectories
                trajdef linear
                        evolve d(t) = 1
```

Figure 4.1.2: IOA Code for Train Example

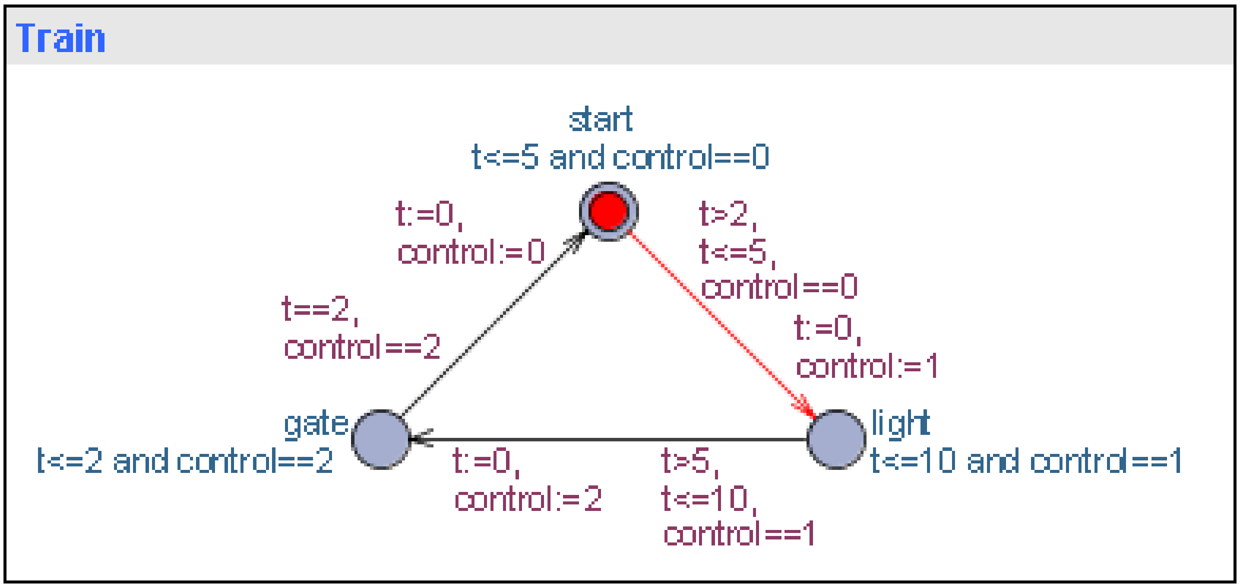


Figure 4.1.3: Train Program in UPPAAL

```
clock time;
int control :=0;
process Train{
  clock t;
  state light{t≤10 and control==1},
    gate{t≤2 and control==2},
    start{t≤5 and control==0};
  init start;
  trans gate → start{guard t==2, control==2; assign t:=0, control:=0; },
    start → light{guard t>2, t≤5, control==0; assign t:=0, control:=1; },
    light → gate{guard t>5, t≤10, control==1; assign t:=0, control:=2; };
}
system Train;
```

Figure 4.1.4: XTA Code for Train Example

## 4.2 UPPAAL-Oriented Restricted Language

UPPAAL is specifically designed to model timed systems, and has most of the features of TIOA built-in. Thus, a few simple restrictions are sufficient to map out a subset of the TIOA language that is translatable into UPPAAL.

### 4.2.1 Variables and Operators

The variables and operators permissible in the ioa2xta translator are also allowed by the tioa2xta translator. These variables and operators are outlined in Sections 3.2.1 and 3.2.2, respectively.

In addition to these variables, variables with a newly created type, *analog*, are available. An *analog* variable models time; it is a clock. In UPPAAL, all clock variables advance with rate one. Thus analog variables can never be assigned non-integer values if they are to be translated into UPPAAL.

The implementation of the type *analog* differs slightly from the TIOA language specification presented in [18]. The current TIOA language specification calls for all real non-continuous variables to be defined with an additional modifier, *discrete*. Since UPPAAL only allows a restricted class of continuous variables, for simplicity, this feature has not been implemented in full. Instead, the new variable *analog* was created to specify the continuous variables representing time.

### 4.2.2 Trajectories

As mentioned in Section 3.2.1, all clocks advance with rate one, and have only integer values. To satisfy these requirements, only trajectories of the form d(t)=1 are allowed.

Urgency and Stopping predicates are also restricted; specifically, only urgency predicates are allowed. Users who wish to translate a program involving stopping conditions must change their program to use urgency predicates. This decision is not based on a restriction in UPPAAL, but is reflective of the way in which TIOA will be translated. Later versions of tioa2xta may include a function that creates urgency predicates out of stopping conditions.

As claimed in [14], stopping conditions are strictly more expressive then urgency predicates. Stopping conditions have the ability to express global halts, where no more transitions can take place after time has stopped evolving. Programs using only urgency predicates have *time reactivity*, meaning that whenever time stops evolving, some transition is enabled. This does not neccesarily mean that more time passes, only that some transition can take place.

Thus restricting TIOA programs to use only urgency predicates ensures that these programs are *time reactive*, and places no other limitations on the program. In addition, [14] points out that changing stopping conditions into urgency predicates is usually possible, and the resulting TIOA is often shorter and more natural, and presents a scheme for doing so.

Taking our Train example, the transition for approaching a crossing might be written with stopping conditions:

```
transitions
   internal approaching
      pre t > 5 ∧ control = light
      eff control := gate, t := 0
...
trajectories
        trajdef linear
                stopCond (control = light ∧  t = 10)
                evolve d(t) = 1
```

The stopping condition can, however, be incorporated into the transition, as an urgency predicate:

```
transitions
   iternal approaching
      pre t > 5 ∧ control = light
      urgent when  t = 10
      eff control := gate, t := 0
...
```

Requiring only the use of urgency predicates is reasonable for translation to UPPAAL, since UPPAAL does not employ stopping conditions. In later versions of the tioa2xta translator, we will consider a feature whereby stopping conditions may be automatically translated into urgency predicates. For the initial version, however, requiring urgency predicates provides a simple and easy solution.

### 4.2.3 Zones of time

Urgency predicates must not include multiple zones of time. Zones of time refer to continuous windows of values for a variable that measures time. An urgency predicate including multiple zones of time might be $t < 5 \setminus / t > 10$, to indicate that the transition is urgent before $t = 5$ or after $t = 10$ but not in between. Such urgency predicates are not allowed in the TIOA to UPPAAL translator. This is based on a restriction in UPPAAL. Multiple zones of time are not permitted in invariant definitions in UPPAAL, therefore we must restrict them from TIOA that will be translated into UPPAAL. [22, 34]

### 4.2.4 Syntax

The above restrictions make it possible to interface TIOA and UPPAAL. The extensions to the IOA language that allow this UPPAAL-oriented restricted sublanguage of TIOA are as follows:

The new variable type defined above in Section 4.2.1, *analog*, is permitted.

Transitions now have an additional, optional field, *urgent when*. This field is for the urgency predicate. When both *pre* and *urgent when* are true, no time can pass until some transition has been taken.

An additional field for trajectories is necessary if any clock variables or urgency predicates are employed. Trajectories are declared in the following manner:

```
trajectories
  trajdef T
    evolve D
```

Here D is a a set of differential or algebraic equation specifying time passage of each clock. Since we restrict this language to clocks of rate 1, D will always be $d(t) = 1$, for *analog* variable $t$.

## 4.3 Toolkit Extension for TIOA

Two parts of the toolkit were extended to accommodate the TIOA additions: the parser and the internal representation of automata.

The parser was extended to allow the additions described in Section 4.2.4. The parser now accepts *analog* as a variable type, *urgent when* as an optional field for a transition definition, and a *trajectories* section with multiple *trajdef* fields. The extended parser generates a Java tree with the new internal automata representation.

The internal representation of timed automata was created by extending the relevant portions of the automaton Java tree. Whenever an extension was made to accommodate TIOA, the resulting class name was pre-pended with a T. For example, the extended Automaton class is called TAutomaton.

Two entirely new classes were created: the Trajectory class, to represent trajectory definitions; and an Urgency class, to represent urgency predicates.

## 4.4 Translation Scheme

The TIOA to UPPAAL translation is modeled after the IOA to UPPAAL translation described in Section 3.3. The major difference in the translation is the use of multiple locations, and the use of location invariants to translate urgency predicates.

### 4.4.1 Variables

As in the ioa2xta translation, all variables in TIOA become variables in UPPAAL. Enumerated variables with $n$ values become integers, with possible values between 1 and $n$.

### 4.4.2 Transitions

Transitions in TIOA become sets consisting of: a single location and a group of transitions that begin at that location. The reason for this is that the urgency predicate of the transition becomes a location invariant in UPPAAL. In the IOA to UPPAAL translation, no transitions were urgent, and thus only a single default location was employed.

It is important to note here that two possible methods for translating TIOA into UPPAAL can be employed: each TIOA transition can map to a location in UPPAAL, or only those transitions with urgency predicates could be mapped to locations, with all other TIOA transitions mapping to UPPAAL transitions originating from a single default location. Both of these translation methods produce valid UPPAAL programs which correctly implement the TIOA program.

The translation method chosen for this thesis maps each TIOA transition to a location in UPPAAL, even if no urgency predicate is declared. A non-declared urgency predicate is assumed to be *urgent when false*, that is, the transition is never urgent. This method was chosen because it makes better use of the graphical features of UPPAAL; users viewing a graphical representation of their program can easily identify locations mapping to transitions from the TIOA program, but the meaning of the default location in the program execution is more ambiguous- it represents several actions taking place. Thus the one-transition one-location translation method was adopted.

The UPPAAL transitions corresponding to the IOA transition have guard and assign clauses as in ioa2xta; the TIOA precondition maps to the guard clause of each transition, and the TIOA effects maps to the assign clause of each transition.

The UPPAAL transitions corresponding to the IOA transition must begin at the start location corresponding to that IOA transition. There can be many of these transitions, one pointing to every other location. However, some of these transitions cannot be ever be taken; if the resulting program state would violate the invariant of the location the transition transitions to, the transition will never fire. In practice, drawing an UPPAAL transition from every location to every other location makes the graphical model cluttered, with many extraneous transitions that can never be taken.

In order to simplify the graphical representation of the program, we eliminate those transitions that are obviously never possible, by comparing the effects clause of the transition with the invariant of the location. This comparisson is done by a helper program, *primitiveSatisfies*, as described in 4.4.3. The pruned graphical representation of the train example is shown in Figure 4.1.3.

### 4.4.3 Pseudo Code

Pseudo code for the translation scheme is presented in Figure 4.4.5.

When the *TimedAutomatonStrip* program initially runs, it creates a map of all possible transitions in the UPPAAL program, and the location they start from. Then, a helper program to the translator, *primitiveSatisfies*, compares simple assignments to real number values with the invariant of all possible end locations. Transitions from the start location to the end location are only added to the UPPAAL process if they pass this check.

## 4.5 Implementation

Implementation of the tioa2xta translator was modeled after the ioa2xta translator. The IOA Toolkit parses an TIOA file describing a Timed Automaton and creates a Java tree with a TAutomaton object at the root.

The uppaal package created for the ioa2xta translator is used as a target for the translation. A parser, TimedAutomatonStrip, walks through the automaton's Java tree and creates a corresponding tree in the uppaal package. Pseudo code for TimedAutomatonStrip is presented in Figure 4.4.5. The XTA file is generated using the printXTA() function; when printXTA() is called on a Project, the function recursively calls printXTA() on all other objects in its tree, and then returns a string that contains the specification of the project in XTA format.

```
TimedAutomatonStrip(TimedAutomaton a){
  Project proj = new Project();
  Project.add(new GlobalClock("time"))
  Process p = new Process();
  for all State s in a {
    Variable v = new Variable(s);
    p.add(v);
  }
  Map transitions = new Map();
  for all Transition t in a {
    Location l = new Location();
    l.setInvariant(t.urgency);
    p.addLocation(l);
    UppaalTransition ut = new UppaalTransition(t);
    ut.setGuard(t.precondition);
    ut.setAssign(t.effects);
    transitions.add(l,ut);
  }
  for all pairs (Location start, UppaalTransition ut) in transitions {
    for all Location end in p.locations {
      if primitiveSatisfies(ut.assign,end.invariant) {
        p.addTransition(start, ut, end)
      }
    }
  }
  proj.add(p);
}

primitiveSatisfies(AssignClause assign, InvariantClause invariant)
  boolean returnValue = true;
  for all assignStatement a in assign {
    if (a.isSimpleAssignment() && a.value.isRealNumber()){
      for all invariantStatement i in invariant {
        if (i.variable = a.variable &&
             (i.value.isRealNumber() && i.value != a.value)){
           returnValue = false;
        } } } }
  return returnValue;
}
```

Figure 4.4.5: Pseudocode for translation from TIOA to XTA

## 4.6 Testing the Translation

The Fischer mutual exclusion algorithm, being a practical time-dependant example, was chosen as a test case for translation. The TIOA code for the Fischer mutual exclusion program is shown in Figure 4.7.12.

The tioa2xta translator outputted an XTA file, shown in Figure 4.7.14. The resulting program was run in the UPPAAL simulator and verifier. As described in Section 4.7.4, the invariants of the Fischer mutual exclusion algorithm were preserved.

## 4.7 Examples

### 4.7.1 Train

The train example is described in Section 4.1. The train program models a train approaching a crossing. The light is signaled, then the gate is lowered for the train to cross (see Figure 4.1.1). Each of these actions take place with a certain urgency. The IOA code for this train is presented in Figure 4.1.2 and the XTA code in figure 4.1.4.

### 4.7.2 Door

The door automaton represents an impatient person ringing a door bell. There are 4 transitions: ringing the bell, the door opening for him to go in, him leaving and resetting, and him getting impatient while waiting and resetting. The time it takes him to ring the doorbell and to get impatient is bounded; those transitions have urgency predicates. The other actions are not urgent.

```
type ControlType = enumeration ready, waiting, entered

automaton Door
        signature
                internal ring, open, impatient, leave
        states
                control := ControlTYpe := ready
                t : analog := 0
        transitions
                internal ring
                        pre t>1 ∧ control = ready
                        urgent when t = 3
                        eff control := waiting, t := 0
                internal open
                        pre control = waiting
                        eff control := entered; t := 0
                internal impatient
                        pre control = waiting
                        urgent when t = 10
                        eff control := ready; t := 0
                internal leave
                        pre control = entered
                        eff control := ready, t:=0
        trajectories
                trajdef linear
                        evolve d(t) = 1
```

Figure 4.7.6: TIOA Code for Door

```
clock time;
process Door{
  int control := 1;
  clock t;
  state
    ring{control==1 and t≤3},
    open{control==2},
    impatient{t≤10 and control==2},
    leave{control==3};
  init ring;
  trans
    ring → ring{guard t>1,control==1; assign control:= 2, t:=0;},
    ring → open{guard t>1,control==1; assign control:= 2, t:=0;},
    ring → impatient{guard t>1,control==1; assign control:= 2, t:=0;},
    ring → leave{guard t>1,control==1; assign control:= 2, t:=0;},
    open → ring{guard control==2; assign control:=3, t:=0;},
    open → open{guard control==2; assign control:=3, t:=0;},
    open → impatient{guard control==2; assign control:=3, t:=0;},
    open → leave{guard control==2; assign control:=3, t:=0;},
    impatient → ring{guard control==2; assign control:=1, t:=0;},
    impatient → open{guard control==2; assign control:=1, t:=0;},
    impatient → impatient{guard control==2; assign control:=1, t:=0;},
    impatient → leave{guard control==2; assign control:=1, t:=0;},
    leave → ring{guard control==3; assign control:=1, t:=0;},
    leave → open{guard control==3; assign control:=1, t:=0;},
    leave → impatient{guard control==3; assign control:=1, t:=0;},
    leave → leave{guard control==3; assign control:=1, t:=0;};
}
system Door;
```

Figure 4.7.7: XTA Code for Door

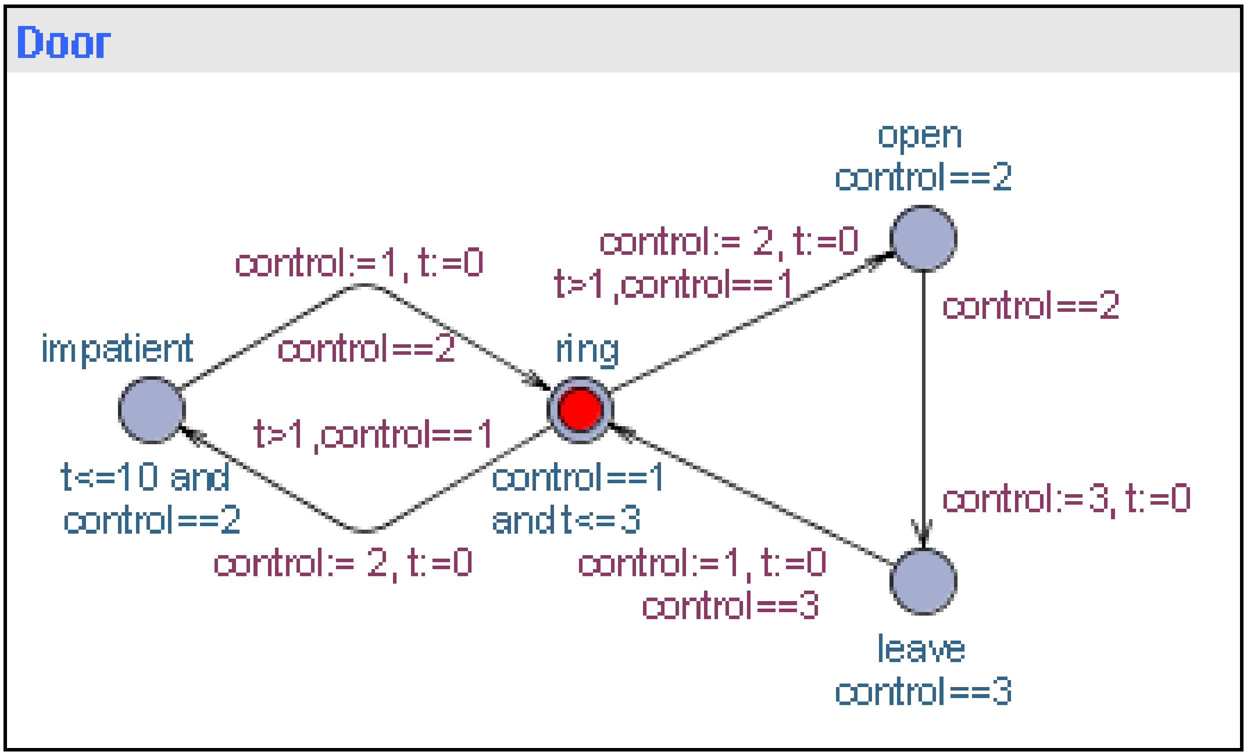


Figure 4.7.8: UPPAAL Representation of the Door Program

```
type PCType = enumeration coming, inplace

automaton SkiLift
        signature
                internal approach, pickup, depart
        states
                pc : PCtype := coming
                t : analog := 0
                passengers : int :=0
        transitions
                internal approach
                        pre pc = coming
                        urgent when t = 5
                        eff pc := inplace; t := 0
                internal pickup
                        pre pc = inplace ∧ passengers <2
                        urgent when t=5
                        eff passengers := passengers+1
                internal depart
                        pre pc=inplace
                        urgent when t=5
                        eff pc := coming; t:=0; p:=0
        trajectories
                trajdef linear
                        evolve d(t) = 1
```

Figure 4.7.9: TIOA Code for Ski Lift

### 4.7.3 Skilift

The Ski Lift Automaton represents a two-seated chair that picks skiers up at the bottom of a mountain. It can fit at most two people, but there are not neccesarily that many riders. In this automaton, the empty chair approaches the pickup location, picks up people either twice, once, or no times, and then departs. The interesting aspect to this automaton is that the clock variable is not reset by the pickup transition; there is a total of 5 seconds for the chair to pickup people and depart, so both transitions rely on the same urgency predicate.

```
clock time;
Process SkiLift{
  int pc := 0,
  clock t,
  int passengers :=0
  state
    approach{t≤5 and pc=0},
    pickup{t≤5 and pc=1 and passengers<2},
    depart{t≤5 and pc=1}
  trans
    approach→approach{guard pc = 0; assign pc := 1, t := 0;},
    approach→pickup{guard pc = 0; assign pc := 1, t := 0;},
    approach→depart{guard pc = 0; assign pc := 1, t := 0;},
    pickup→approach{guard pc = 1,passengers <2;
                     assign passengers := passengers+1;},
    pickup→pickup{guard pc = 1,passengers <2;
                   assign passengers := passengers+1;},
    pickup→depart{guard pc = 1,passengers <2;
                   assign passengers := passengers+1;},
    depart→approach{guard pc=1; assign pc := 0, t:=0,p:=0;},
    depart→pickup{guard pc=1; assign pc := 0, t:=0,p:=0;},
    depart→depart{guard pc=1; assign pc := 0, t:=0,p:=0;};
}
system SkiLift
```

Figure 4.7.10: XTA Code for Ski Lift

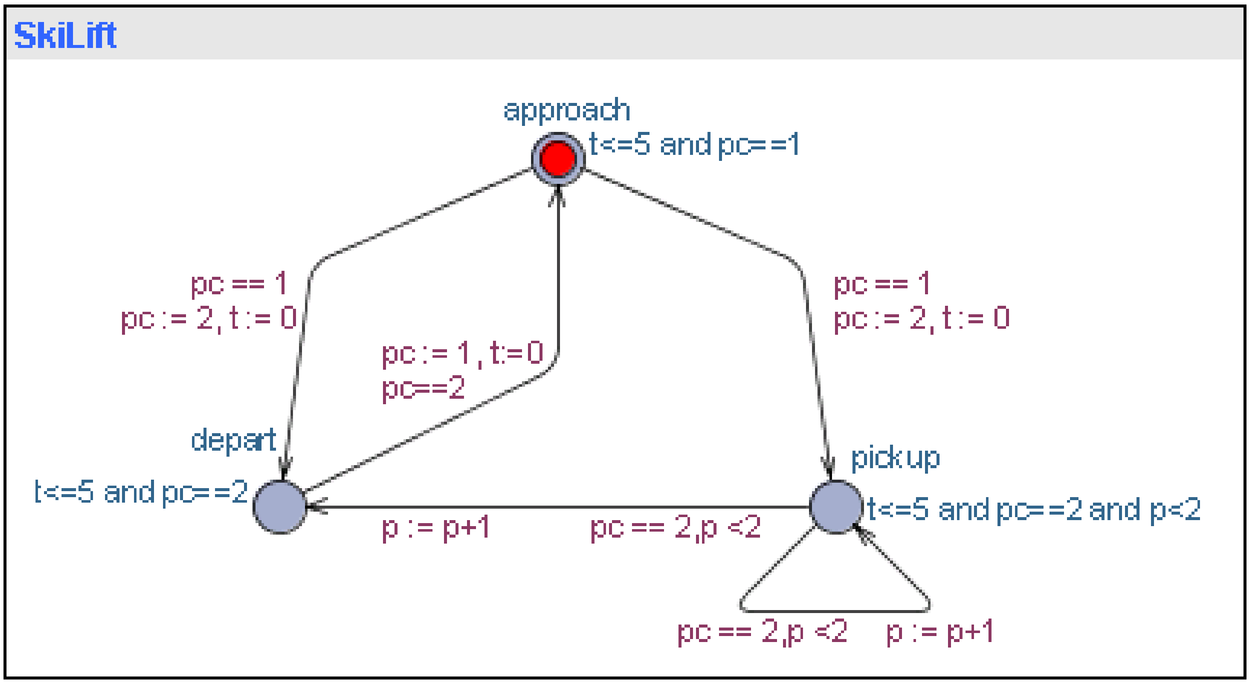


Figure 4.7.11: UPPAAL Representation of the Ski Lift Program

### 4.7.4 Fischer Mutual Exclusion

The Fischer Mutual Exclusion algorithm ensures mutual exclusion using timing constraints and a shared variable, with time bounds on the exclusion period. This protocol has been studied and verified using UPPAAL in previous case studies [24], and provides a good example of a time-dependent algorithm.

Here we show the Fischer algorithm for two processes, adapted from the Fischer algorithm for $n$ processes in 4. This example has been adapted to use urgency predicates rather then stopping conditions. In addition, the upper and lower time bounds have been hard coded with values of 5 seconds and 3 seconds, respectively. The TIOA code is shown in Figures 4.7.12 and 4.7.13, and the XTA translation is shown in 4.7.14.

Unlike the previous examples, the Fischer algorithm was not initially hand translated before it was automatically translated. Similarly to the use of the Peterson Exclusion Algorithm for the ioa2xta translation, the Fischer algorithm was reserved as a test program to see if the tioa2xta translation scheme developed by hand-translating the previous three programs was successful.

The Fischer Mutual Exclusion algorithm was simulated in UPPAAL, and the properties in Figure 4.7.16 were verified using the UPPAAL verification tools. See Section 2.3.5 and corresponding Figure 2.3.7 for a description of the verification syntax. The verification of these properties serves as a successful test of the translator.

```
type pcType = enumeration of rem, test, set,
            check, leavetry, crit, leaveexit

automaton FischerME
  signature
    internal try1, crit1, exit1, rem1, test1, set1, check1, reset1,
      try2, crit2, exit2, rem2, test2, set2, check2, reset2

  states
    x : int := 0,
    pc1 : pcType := rem,
    pc2 : pcType := rem,
    lastset1 : int,
    lastset2 : int,
    firstcheck1 : int,
    firstcheck2 : int,
    t : analog := 1

  trajectories
    trajdef linear
      evolve d(t) = 2
```

Figure 4.7.12: TIOA Code for Fischer Algorithm (continued in Figure 4.7.13)

```
transitions
  internal try1
    pre pc1 = rem
    urgent when t = lastset1
    eff pc1 := test
  internal crit1
    pre pc1 = leavetry
    urgent when t = lastset1
    eff pc1 := crit
  internal test1
    pre pc1 = test; x=0
    urgent when t = lastset1
    eff pc1 := set;
        lastset1 := t + 5
  internal exit1
    pre pc1 = crit
    urgent when t = lastset1
    eff pc1 := reset
  internal set1
    pre pc1 = set
    urgent when t = lastset1
    eff x := 1; pc1 := check;
      lastset1 := 0;
      firscheck1 := t + 3
  internal reset1
    pre pc1 = reset
    urgent when t = lastset1
    eff x := 0;
        pc1 := leaveexit
  internal check1a
    pre pc1 = check ∧
      firstcheck1 < t ∧
      x = 1
    urgent when t = lastset1
    eff pc1 := leavetry
  internal check1b
    pre pc1 = check ∧
      firstcheck1 < t ∧
      x ≠ 1
    urgent when t = lastset1
    eff pc1 := test
  internal rem1
    pre pc1 = leaveexit
    urgent when t = lastset1
    eff pc1 := rem
```

```
  internal try2
    pre pc2 = rem
    urgent when t = lastset2
    eff pc2 := test
  internal crit2
    pre pc2 = leavetry
    urgent when t = lastset2
    eff pc2 := crit
  internal test2
    pre pc2 = test; x=0
    urgent when t = lastset2
    eff pc2 := set;
        lastset2 := t + 5
  internal exit2
    pre pc2 = crit
    urgent when t = lastset2
    eff pc2 := reset
  internal set2
    pre pc2 = set
    urgent when t = lastset2
    eff x := 2; pc2 := check;
      lastset2 := 0;
      firscheck2 := t + 3
  internal reset2
    pre pc2 = reset
    urgent when t = lastset2
    eff x := 0;
        pc2 := leaveexit
  internal check2a
    pre pc2 = check ∧
      firstcheck2 < t ∧
      x = 2
    urgent when t = lastset2
    eff pc2 := leavetry
  internal check2b
    pre pc2 = check ∧
      firstcheck2 < t ∧
      x ≠ 2
    urgent when t = lastset2
    eff pc2 := test
  internal rem2
    pre pc2 = leaveexit
    urgent when t = lastset2
    eff pc2 := rem
```

Figure 4.7.13: TIOA Code for Fischer Algorithm Begun in Figure 4.7.12

```
clock time;
Process FischerME{
  int x := 0;
  int pc1 := rem;
  int pc2 := rem;
  int lastset1;
  int lastset2;
  int firstcheck1;
  int firstcheck2;
  clock t;
  state
    try1{t<lastset1}, crit1{t<lastset1}, test1{t<lastset1},
    exit1{t<lastset1}, set1{t<lastset1}, reset1{t<lastset1},
    check1a{t<lastset1}, check1b{t<lastset1}, rem1{t<lastset1},
    try2{t<lastset2}, crit2{t<lastset2}, test2{t<lastset2},
    exit2{t<lastset2}, set2{t<lastset2}, reset2{t<lastset2},
    check2a{t<lastset2}, check2b{t<lastset2}, rem2{t<lastset2};
```

Figure 4.7.14: XTA Code for Fischer Mutual Exclusion Algorithm

```
  trans
    try1→test1{guard pc1 = rem; assign pc1 := test;},
    crit1→reset1{guard pc1 = leavetry; assign pc1 := crit;},
    test1→set1{guard pc1 = test, x=0;
                assign pc1 := set, lastset1 = t + 5;},
    exit1→reset1{guard pc1 = crit; assign pc1 := reset;},
    set1→check1a{guard pc1 = set; assign x := 1, pc1 := check,
                lastset1 := 0, firscheck1 := t + 3;},
    set1→check1b{guard pc1 = set; assign x := 1, pc1 := check,
                lastset1 := 0, firscheck1 := t + 3;},
    reset1→rem1{guard pc1 = reset;
                 assign x := 0, pc1 := leaveexit;},
    check1a→crit1{guard pc1 = check, firstcheck1 < t, x = 1;
                assign pc1 := leavetry;},
    check1b→set1{guard pc1 = check, firstcheck1 < t, x != 1;
                assign pc1 := test;},
    rem1→test1{guard pc1 = leaveexit; assign pc1 = rem;},
    try2→test2{guard pc2 = rem; assign pc2 := test;},
    crit2→reset2{guard pc2 = leavetry; assign pc2 := crit;},
    test2→set2{guard pc2 = test, x=0;
                assign pc2 := set, lastset2 = t + 5;},
    exit2→reset2{guard pc2 = crit; assign pc2 := reset;},
    set2→check2a{guard pc2 = set; assign x := 2, pc2 := check,
                lastset2 := 0, firscheck2 := t + 3;},
    set2→check2b{guard pc2 = set; assign x := 2, pc2 := check,
                lastset2 := 0, firscheck2 := t + 3;},
    reset2→rem2{guard pc2 = reset;
                 assign x := 0, pc2 := leaveexit;},
    check2a→crit2{guard pc2 = check, firstcheck2 < t, x = 2;
                assign pc2 := leavetry;},
    check2b→set2{guard pc2 = check, firstcheck2 < t, x != 2;
                assign pc2 := test;},
    rem2→test2{guard pc2 = leaveexit; assign pc2 = rem;};
}
system FischerME;
```

Figure 4.7.15: XTA Code for Fischer Algorithm Continued from Figure 4.7.14

$E <> \ x = 1:$ for some execution path, process 1 gets a turn
$E <> \ x = 2:$ for some execution path, process 2 gets a turn
$A[\,]\ pc1 \ = \ test \ --> \ x = 1:$ if process 1 is testing, it will eventually get a turn
$A[\,]\ pc2 \ = \ test \ --> \ x = 2:$ if process 2 is testing, it will eventually get a turn
$A[\,]\ pc1 \ = \ crit$ impy $x = 1:$ if process 1 is critical, it gets a turn
$A[\,]\ pc2 \ = \ crit$ impy $x = 2:$ if process 2 is critical, it gets a turn
$A[\,]\ lastset1 <= t + 5:$ process 1 gets no more then 5 seconds
$A[\,]\ lastset2 <= t + 5:$ process 2 gets no more then 5 seconds
$A[\,]\ firstcheck1 >= t + 3:$ process 1 gets at least 3 seconds
$A[\,]\ firstcheck2 >= t + 3:$ process 2 gets at least 3 seconds

Figure 4.7.16: Properties of the Fischer Mutual Exclusion Program

# Chapter 5

# Translating Composite Automata into UPPAAL

## 5.1 Overview

The previous chapters have presented automatic translations of primitive automata from IOA and TIOA to UPPAAL. This chapter discusses a proposed method for translating composite automata, using the tioa2xta translator, and features of UPPAAL.

## 5.2 Method

Multiple Timed I/O Automata are composed together by matching the input transitions of one automaton with the output transitions of another. [20]

Each automaton in the composition will be modeled as a process in UPPAAL. The composed automata will then become an UPPAAL project containing these processes and the UPPAAL synchronization variables that are generated to connect transitions between processes.

When dealing with a composition of composite automata, there is a question of how to represent the programs in UPPAAL. For instance, when composing a closed pair of composite automata with another, third automaton, there are two possible representations: all three automata can become separate processes, or the closed pair of automata can be translated into a primitive automaton and represented as a single process. In more complicated systems, the issue of representation becomes important for visualizing levels of abstraction in a program. In developing a translator for composed automata, some mechanism to provide a measure of control to users should be defined.

### 5.2.1 Synchronization Channels

UPPAAL supports synchronization of multiple processes using channels. A special type in UPPAAL, *chan*, represents a channel variable. These channel variables can be used to synchronize two or more transitions in different processes. This is accomplished by setting the *sync* field of a transition to the desired channel variable.

When two or more transitions share a channel, they must fire at the same time. If any channel does not have its guard clause (precondition) met, none of the transition sharing that channel can take place.

In this way, the output transition of an automaton can be linked with the input transitions of other automata. The output action of one automaton, represented by a transition in UPPAAL, must fire at the same time as the transitions that represent the corresponding input actions of the other automata.

## 5.3 Translation Scheme

In order to translate a TIOA program into an UPPAAL specification, the XTA format should again be used. This takes advantage of the existing tioa2xta translator described in Section 4.

Throughout this section, it may be helpful to refer to the Channels and Nodes composition example in Section 5.4.

Initially, each individual automaton should be translated using tioa2xta into a UPPAAL-

Java tree, with a Process at the root. The uppaal package for the IOA Toolkit is described in Section 3.4.1. This Process represents all information from the automaton. In the Channels and Nodes example, each node and each channel becomes a process; see Figure 5.4.4.

To compose the automata transitions representing matching actions, where the output transition of one automaton matches the input action of another, must share a synchronization variable.

Thus, for set of transitions where one automaton's output transition matches input transitions of other automata, generate a global synchronization variable, to be declared in the parent Project. In choosing variable names, it might be good to use names that begin with "sync." In the Channels and Nodes example, these variables are called "syncA", "syncB", etc, for uniformity and easy reading. Synchronization variables are declared at the start of XTA files, with all the other variables, such as the global clock:

```
clock time;
chan syncA;
chan syncB;
...
```

According to the translation scheme for the individual Timed I/O Automata, each action taken by the automata will correspond to a location and a set of transitions in UPPAAL. To each of these transitions, the synchronization variable needs to be added as a channel. For example:

```
...
  process NodeA{
  ...
  trans
    send → receive{
      guard queued == true;
      sync syncA!;
      assign queued := false};
}
...
```

Synchronization variables are used by transitions with the syntax *channelName*! or *channelName*?, representing the transition speaking on that channel (!), or listening (?). Only one transition can speak on a channel, but multiple transitions can listen. This corresponds exactly to the notion in IOA that one automatons output action can correspond to multiple automatas input actions. Thus the output transition is the speaker, and the input transitions are the listeners. The effect of the synchronization variable is to force all these transitions to happen at once. The transitions may only take place if all the preconditions are met. According to IOA specification, input transitions are always enabled, and have no preconditions. Since only one transition will correspond to the output, only that transition can have preconditions.

In the UPPAAL-Java tree, the composition with synchronization variables should be accomplished by creating a new synchronization variable in each transition. As in the tioa2xta translation, the printXTA() function will take care of output details for the entire project.

## 5.4 Channels and Nodes Example

This example consists of a pair of nodes, communicating through two one-way channels. This example was chosen to demonstrate composition because of its classic nature, and ready application to distributed processes. The XTA code presented in Figure 5.4.5 was hand-translated from the TIOA code in Figures 5.4.2 and 5.4.1, and represents the proposed automatic translation scheme.

It is important to observe that although each node and each channel has the same TIOA code, each becomes a separate process when instantiated. Each node and each channel in the system becomes a process, for four processes total, see Figure 5.4.4. The resulting XTA code is still one program, since it represents the entire composed project, see Figure 5.4.5.

Following along with Figures 5.4.4 and 5.4.3, the nodes are NodeA and NodeB, and the channels are Channel1 and Channel2. NodeA sends a message through Channel1, so in the IOA composition, the *send* output transition of NodeA must connect to the *send* input transition of Channel1. Thus, in the UPPAAL representation, these transitions are connected with a synchronization variable, *syncA*. The send transition of NodeA sets *syncA!*, and the send transition of Channel1 listens on the same channel with *syncA?*.

```
automaton Channel
  signature
    input send
    output receive
    internal wait
  states
    t : analog := 0,
    intransit: boolean := false
  transitions
    input send
      eff intransit := true
    output receive
      pre intransit
      urgent when t = 5
      eff intransit := false ; t := 0
    internal wait
      pre ¬intransit
  trajectories
    trajdef linear
      evolve d(t) = 1
```

Figure 5.4.1: TIOA Code for a Channel

For these synchronized transitions fire, all preconditions must be met, that, NodeA.*queued*$==$ $true$, and all effects must occur, that is, Channel1.*intransit*$==$ $true$, NodeA.*queued*$:=$ $false$, NodeA.*ready*$:=$ $true$, and NodeA.$t$$:=$ 0.

Similarly, Channel1s output action *receive* must connect to NodeBs input action *receive*, completing the communication from NodeA to NodeB. Thus, in UPPAAL, these transitions are synchronized with *syncB*. The *receive* transition of Channel1 sets *syncB!*, and the *receive* transition of NodeB is synchronized with *syncB?*. All preconditions of these transitions must be met for them to fire.

```
automaton Node
  signature
    input receive
    internal compute
    output send
    internal wait
  states
    t : analog := 0,
    ready: boolean := true,
    queued: boolean := false
  transitions
    input receive
      pre ready
      eff ready := false
    internal compute
      pre ¬ready
      urgent when t = 10
      eff queued := true
    output send
      pre queued
      urgent when true
      eff queued := false ; ready := true ; t := 0
    internal wait
      pre ready
  trajectories
    trajdef linear
      evolve d(t) = 1
```

Figure 5.4.2: TIOA Code for a Node

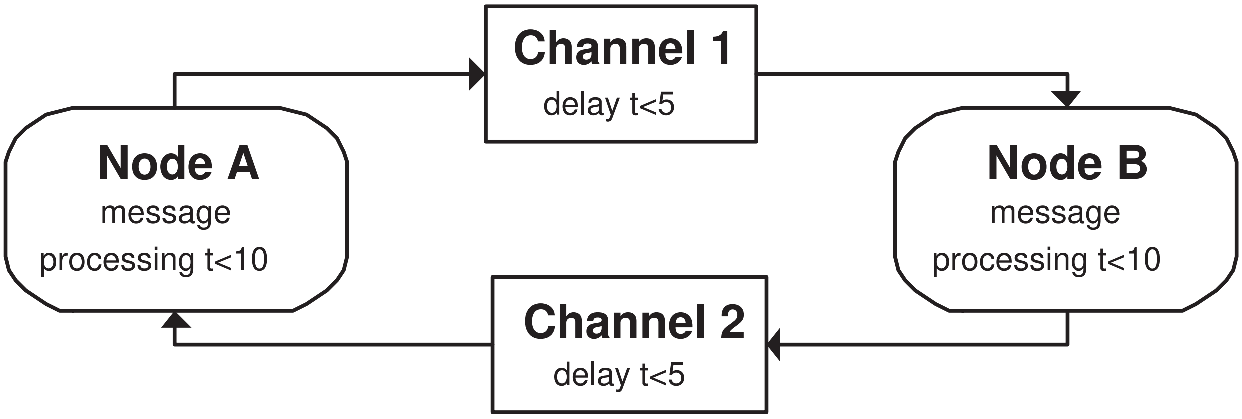


Figure 5.4.3: Two Nodes Communicating Via Two One-way Channels

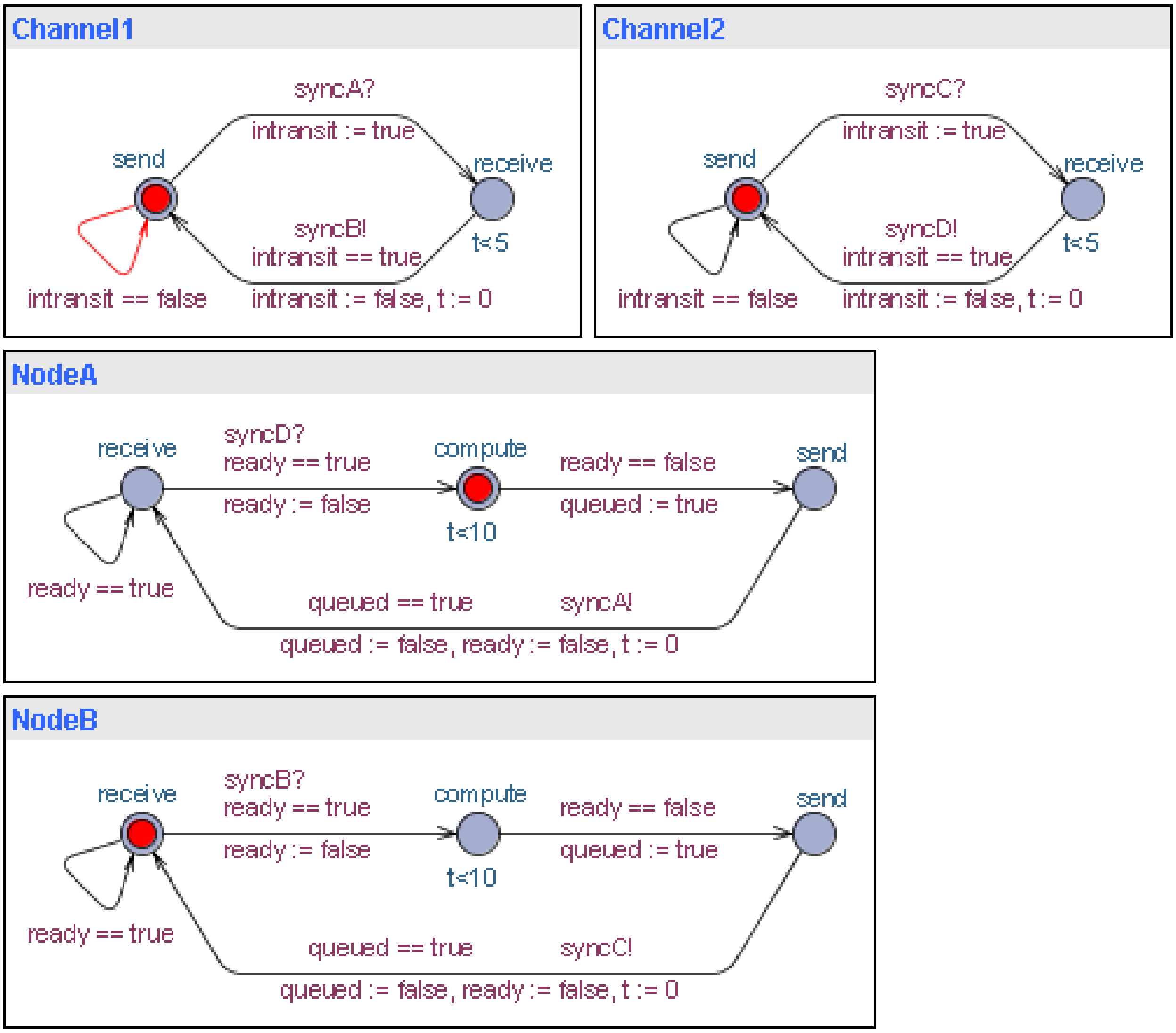


Figure 5.4.4: Composed Channels and Nodes in UPPAAL

```
clock time;  chan syncA;  chan syncC;
             chan syncB;  chan syncD;

process Channel1{
    clock t;
    bool intransit := false;
  state receive{t<5}, send;
    init send;
  trans
    send → receive{sync syncA?;
      assign intransit := true; },
    receive → send{
      guard intransit == true;
      sync syncB!;
      assign intransit := false,
        t := 0; },
    send → send{
      guard intransit == false; };
}
process Channel2{
    clock t;
    bool intransit := false;
  state receive{t<5}, send;
    init send;
  trans
    send → receive{sync syncC?;
      assign intransit := true; },
    receive → send{
      guard intransit == true;
      sync syncD!;
      assign intransit := false,
        t := 0; },
    send → send{
      guard intransit == false; };
}
process NodeA{
    clock t;
    bool ready := false;
    bool queued := false;
  state receive, send,
      compute{t<10};
    init compute;
  trans
    receive → compute{
      guard ready == true;
      sync syncD?;
      assign ready := false; },
    compute → send{
      guard ready == false;
      assign queued := true; },
    receive → receive{
      guard ready == true; },
    send → receive{
      guard queued == true;
      sync syncA!;
      assign queued := false,
        ready := false, t:= 0; };
}
process NodeB{
    clock t;
    bool ready := true;
    bool queued := false;
  state receive, send,
      compute{t<10};
    init receive;
  trans
    receive → compute{
      guard ready == true;
      sync syncB?;
      assign ready := false; },
    compute → send{
      guard ready == false;
      assign queued := true; },
    receive → receive{
      guard ready == true; },
    send → receive{
      guard queued == true;
      sync syncC!;
      assign queued := false,
        ready := false, t := 0; };
}
system Channel1, Channel2, NodeA, NodeB;
```

Figure 5.4.5: XTA Code for Composed Nodes and Channels

# Chapter 6

# Conclusions

## 6.1 Summary

This thesis presents a first-stage implementation of Timed IOA in the IOA Toolkit, to assist with the development of timed systems. These extensions work for primitive automata that have clock variables evolving at the same rate as real-time. This thesis lays the framework for a complete extension of the toolkit, which would allow a broader range of automata, including composed automata to be simulated. A scheme for translating composed automata into UPPAAL is also presented.

This thesis also includes a translator from IOA and TIOA into UPPAAL, allowing the simulation of IOA and TIOA programs in an easy-to-use interface, and the modeling of time and time constraints. The project has connected two major toolsets for developing and verifying distributed systems, thereby broadening the use of both.

The tools developed in this thesis project are particularly useful to users as a first-step check when designing time-dependant distributed systems, allowing them to catch conceptual errors early. This thesis makes it possible to verify time-related safety properties expressed in the model checking language of UPPAAL.

## 6.2 Future Work

### 6.2.1 Toolkit Extensions

Now that the TIOA language has been formalized, it is time to extend the IOA Toolkit in its entirety to allow timed automata. Translating composed Timed I/O Automata into UPPAAL is the next logical step in this project. The work by Josh Tauber on composition [38] would be a useful reference for this extension.

Aside from the issue of translating more complex programs into UPPAAL, extending the composer is a necessary step towards extending the IOA Simulator to allow timed automata. Further extensions should focus on broadening the functions and variables available under TIOA.

The end goal should be a new TIOA Toolkit, which incorporates all the features of the IOA toolkit, and works for timed automata.

### 6.2.2 Implementing Stopping Conditions

The current TIOA implementation is limited to urgency predicates as a means for specifying time bounds. Urgency predicates are useful in localizing upper-bound conditions on time passage by associating the timing constraints with individual transitions. On the other hand, stopping conditions are better suited to the specification of global constraints on time passage. Although in most examples that arise, urgency predicates are sufficient to specify timing constraints, there will be some cases where stopping conditions are necessary, as outlined in Section 4.2.2, and there may be some cases where stopping conditions are simply prefered.

specify global While urgency predicates allow a wide range of programs to be simulated, and provide very few practical limitations, some users may prefer to use stopping conditions. Stopping conditions can be used to more easily specify program-wide halts, for example, since urgency predicates localize stopping constraints whereas stopping conditions specify global, Extending the Trajectory definition in the IOA Toolkit's TIOA extension to allow stopping conditions should be a future goal of this project. While it is not clear how stopping conditions can be translated into UPPAAL, they should be implemented in the TIOA Toolkit, to provide more complete flexibility of the language.

### 6.2.3 TAME and PVS Interface for TIOA

The Timed Automata Modeling Environment, TAME [1], is a project developed at the US Naval Research Laboratory by Myla Archer. TAME provides an interface to the PVS theorem prover for Timed I/O Automata [1]. Interfacing the TIOA Toolkit with TAME would put the power of the PVS theorem prover at the disposal of distributed systems developers using the TIOA Toolkit.

This project should be pursued in parallel with the extension of the IOA Toolkit to TIOA. By using the requirements of TAME as a guide, the important features of the TIOA Toolkit can be isolated, and developed first.

As this thesis has indicated, it's best to start small, with the minimum possible requirements, and expand from there.

# Bibliography


[1] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. *UITP*, July 1998.

[2] Johan Bengtsson, W. O. David Griffioen, Kare J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using uppaal. In *8th International Conference on Computer-Aided Verification*, pages 244–256, New Brunswick, New Jersey, July 31-3 August 1996.

[3] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, New Jersey, 22-24 October 1995.

[4] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2001.

[5] Anna E. Chefter. A simulator for the IOA language. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 1998.

[6] Laura Dean. Improved simulation of input/output automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2001.

[7] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington Department of Computer Science and Engineering, (Seattle, Washington), 2000.

[8] Michael D. Ernst. Summary of dynamically discovering likely program invariants. In *ICSM 2001: Proceedings of the International Conference on Software Maintenance*, pages 540–544, Florence, Italy, November 6-10 2001.

[9] Michael D. Ernst and et all. *Daikon Invariant Detector User Manual: Daikon version 3.0.4*. MIT Computer Science and Artificial Intelligence Laboratory, 1 May 2004.

[10] Stanislav Funiak. *Model Checking IOA Programs with TLC*. MIT Computer Science and Artificial Intelligence Laboratory, 2001.

[11] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[12] Stephen J. Garland, Nancy A. Lynch, Joshua A. Tauber, and Mandana Vaziri. *IOA: User Guide and Reference Manual*. MIT Computer Science and Artificial Intelligence Laboratory, 2003.

[13] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Formal Language for Input/Output Automata*. MIT Computer Science and Artificial Intelligence Laboratory, 1997.

[14] Biniam Gebremichael and Frits Vaandrager. *On Specifying Urgency in Timed I/O Automata*. Nijmegen Institute for Computing and Information Sciences, 2004.

[15] Klaus Havelund, Kim G. Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker uppaal. In *5th International AMAST Workshop on Real-Time and Probabilistic Systems*, volume Lecture Notes in Computer Science Vol. 1601, pages 277–298, 1999.

[16] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using uppaal. In

*18th IEEE Real-Time Systems Symposium*, pages 2–13, San Francisco, California, 3-5 December 1997.

[17] Thomas Hune, Kim G. Larsen, and Paul Pettersson. LEGO MIND STROMS systems - control program synthesis and verification of rcxtm systems. In *VHS Deliverable CS.5.2: Report on VHS Case Study 5 Sidmar Steel Plant at Ghent, Belgium*, 1999.

[18] Dilsun K. Kaynar, Nancy Lynch, Sayan Mitra, and Christine Robson. *Design for TIOA Modeling Language*. MIT Computer Science and Artificial Intelligence Laboratory, 12 April 2004.

[19] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS 2003: The 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

[20] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata: Technical Report MIT-LCS-TR-917*. MIT Computer Science and Artificial Intelligence Laboratory, 2004.

[21] Dilsun Kirli Kaynar, Anna Chefter, Laura Dean, Stephen J. Garland, Nancy A. Lynch, Toh Ne Win, and Antonio Ramirez-Robredo. Simulating nondeterministic systems at multiple levels of abstraction. In *CONCUR 2002 Tools Day*, Brno, Czech Republic, August 4, 2002.

[22] Kim G. Larsen. Formal methods for real time systems: Automatic verification & validation. In *ARTES Summer School*, August 1998.

[23] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic model-checking for real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, New Jersey, 22-24 October 1995.

[24] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *10th International Conference on Fundamentals of Computation Theory*, pages 62–88, Dresden, Germany, 22-25 August 1995.

[25] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *In Springer International Journal of Software Tools for Technology Transfer*, pages 1(1+2), 1997.

[26] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In *FORTE '94: Seventh International Conference on Formal Description Techniques*, pages 259–273, Berne, Switzerland, October 4–7, 1994. Chapman & Hall.

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

[28] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, MIT Computer Science and Artificial Intelligence Laboratory, April 1987.

[29] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. In *CWI-Quarterly*, pages 2(3):219–246, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, September 1989.

[30] Toh Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2003.

[31] Toh Ne Win, Michael D. Earnst, Stephen J. Garland, Dilsun Kaynar, and Nancy A. Lynch. *Using Simulated Execution in Verifying Distributed Algorithms.* MIT Computer Science and Artificial Intelligence Laboratory.

[32] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kirli, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. In *VMCAI*, New York, January 9–11, 2003.

[33] Tsvetomir P. Petrov, Anya Pogosyants, Stephen J. Garland, Victor Luchangco, and Nancy A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Application, and Tools (FORTE/PSTV)*, pages 29–44, Kaiserslautern, Germany, October 8–11, 1996. Chapman & Hall.

[34] Paul Pettersson and Kim G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, 2000.

[35] J. Antonio Ramrez-Robredo. Paired simulation of I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2000.

[36] Jørgen F. Søgaard-Anderson, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anya Pogosyants. Computed-assisted simulation proofs. In Costas Courcoubetis, editor, *Fifth Conference on Computer-Aided Verification (CAV '03)*, volume 697 of *LNCS*, pages 305–319, Elounda, Greece, June 1993. Springer.

[37] Edward Solovey. Simulation of composite I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2003.

[38] Joshua A. Tauber. Verifiable code generation from abstract I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2004. In progress.

[39] Michael J. Tsai. Code generation for the IOA language. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2002.