TOWARDS MODEL CHECKING BSV IN UPPAAL

August 2013

Hörður Hauksson
Master of Science in Computer Science

TOWARDS MODEL CHECKING BSV IN UPPAAL

Hörður Hauksson
Master of Science
Computer Science
August 2013
School of Computer Science
Reykjavík University

M.Sc. RESEARCH THESIS

ISSN 1670-8539

Towards model checking BSV in Uppaal

by

Hörður Hauksson

Research thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in **Computer Science**

August 2013

Research Thesis Committee:

Anna Ingólfsdóttir, Supervisor
Professor, Reykjavik University

Marjan Sirjani
Professor, Reykjavik University

Luca Aceto
Professor, Reykjavik University

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this research thesis entitled **Towards model checking BSV in Uppaal** submitted by **Hörður Hauksson** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

_____

Date

_____

Anna Ingólfsdóttir, Supervisor
Professor, Reykjavik University

_____

Marjan Sirjani
Professor, Reykjavik University

_____

Luca Aceto
Professor, Reykjavik University

Towards model checking BSV in Uppaal

Hörður Hauksson

August 2013

Abstract

In recent years the complexity of hardware design for parallel processes has increased considerably. Despite advances in structural abstractions of description languages none has the ability to simulate reactive systems and model check in a graphical manner. In this thesis we argue for the viability of translating between two seemingly different languages, the high level hardware description language Bluespec SystemVerilog and the modelling language of Uppaal, an integrated environment for validating and verifying real time systems. We propose a scheme for mapping constructs between the two languages and provide a prototype of a translator from a shallow version of BSV to Uppaal.

Skilgreiningamál sem net tímaháðra stöðuvéla

Hörður Hauksson

Ágúst 2013

Útdráttur

Hönnun smárása hefur á síðari árum orðið sífellt flóknari. Líkön fyrir samsíða keyrslur hafa reynst sérlega erfið viðureignar og hönnunargalla hefur oftar en ekki fyrst orðið vart á framleiðslustigi rása eða við notkun þeirra. Þrátt fyrir framfarir í gerð skilgreiningamála gefur ekkert þeirra færi á myndrænni framsetningu við að greina líkön. Í þessari ritgerð leiðum við rök að því að þýða megi á milli skilgreiningamálsins Bluespec SystemVerilog (BSV) og Uppaal, sem er tæki til að sannreyna magþráða kerfi á myndrænan hátt og finna veikleika í hönnun þeirra. Sýnt er fram á að þýða megi milli þessara mála og sett er fram frumgerð að þýðanda sem þýðir úr einfaldaðri útgáfu af BSV yfir í líkanamál Uppaal.

To
María Dóra Hrafnsdóttir
and
Kári Hrafnsson
who also contributed
a little bit

Acknowledgements

I want to thank the international group of academics and fellow students at Reykjavik University for concurrently sharing a small but significant part of their lives throughout my study. Special thanks to my supervisor Anna for all the encouragement and Luca for making me believe that theoretical computer science is easy to understand.

Contents

List of Figures

Listings

Chapter 1

Introduction

In the scientific literature several formalisms have been proposed for studying the behaviour of concurrent systems. Some are specialized with emphasis on hardware components, but others have a graphical interface and are better suited for simulation and model checking.

Of the former type is Bluespec SystemVerilog (BSV) [1] a programming language that can be compiled into hardware code (synthesizable) and is used in the design of electronic systems. BSV is also equipped with a standard compiler that compiles into executable code.

A commonly used level of abstraction in hardware design is register transfer level (RTL), where hardware is viewed as a combination of registers and combinational logical devices. Registers are the elements in the circuit that have memory properties and combinational logical devices describe how the values change over time. The values of these stateful elements at any given point in time determine the state of the system.

The updates of the registers are designed to be performed simultaneously, resulting in faster hardware. In concurrent systems multiple processes are running in parallel and designers have to address problems such as deadlocking and resource starvation. These problems have proved to be hard to solve and become increasingly harder as the systems grow in size.

Hardware description languages are a means by which to design the hardware so that these updates are made in the most dependable and efficient way. The designs are converted directly into hardware descriptions that can then be converted into hardware. Before the actual hardware is created it is desirable to be able to reason formally about how the hardware described in a hardware description language will behave.

Hardware execution of a program is done on the bit level and we are interested in how the bits flow between the stateful elements of the circuit. The logical devices are comprised of transistors which perform the desired logic. A clock is responsible for advancing the time in the system and it is beneficial to perform as many bit operations in each clock cycle as possible. In each cycle the state is changed based on how the previous state, the input, and how the combinational elements are connected.

The execution model for hardware is considerably different from the traditional model for CPU. The aim of the design process is to create circuits which perform specialized tasks using highly parallel calculations, and to describe how the execution of those integrated circuit (IC) will be performed.

As with other programming languages, the evolution of hardware languages has been to abstract away from underlying low level instructions and form high level constructs such as modules or classes. This has led to a shift in paradigm and the hardware design procedure obeys the same rules as software development in many ways. Basic low level elements are hidden from the designer and instead of working with elements of a circuit (s)he can focus on the overall behaviour of the system. Another advantage of this approach is that behavioural problems in the design can be discovered at an earlier stage with external tools that can verify and check system models.

BSV

BSV is a hardware description language with high level constructs. It was developed at Massachusetts Institute of Technology[1] in association with Bluespec Inc[2]. Originally it was a technology for synthesizing from Term Rewriting Systems [5] but has developed into the high level description language it is today.

BSV is an object oriented language where modules are the main building blocks. Higher level abstraction facilitates modular designs where modules can be aggregated to form complex systems with submodule hierarchy.

The communication between modules is expressed using rule based interface methods which allow rules to be composed from fragments that span module boundaries.

Concurrency in BSV is based on atomic guarded rules that run in parallel. Because rules are atomic, they eliminate a majority of problems with concurrent design such as race con-

[1] http://www.mit.edu/
[2] http://www.bluespec.com/synthesizable-models.html

ditions. Other widely used hardware languages at the register transfer level are Verilog[3], VHDL[4] and SystemC[5].

Hardware constructions are static so BSV is statically typed, meaning that the type of a variable is determined at compile time. Each and every variable and every expression is of a given type. Variables can only be assigned values which have compatible types. Special hardware types (values carried on wires, stored in registers, fifos or memories, etc.) [1, p. 36] determine the state of a BSV model.

Along with the compiler that creates hardware code, a standard compiler exists that compiles programs into executable code. Bluespec SystemVerilog can therefore also be viewed as a full blown programming language.

Uppaal

A formalism better suited for simulation and model checking is Uppaal [2]. It is rooted in timed automata theory [6]. More precisely systems are modeled in Uppaal as networks of timed automata, extended with data types.

Uppaal is an integrated environment for modeling, validating and verifying models. A description language is compiled into a graphical representation (Figure 1.1) so that the behaviour of models can be analyzed in a graphical user interface.

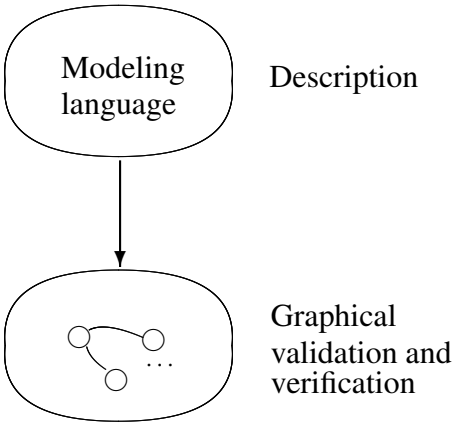


Figure 1.1: Compilation to a graphical representation

[3] http://www.verilog.com/
[4] http://www.vhdl.org/
[5] http://www.systemc.org/

Uppaal is widely used in industry for checking correctness of various hardware and software designs.

Uppaal is developed in collaboration between the Department of Information Technology at Uppsala University[6], Sweden and the Department of Computer Science at Aalborg University[7] in Denmark. Two versions of the environment are available, one for commercial applications and one for work performed by researchers or students at academic institutions. Several extensions of the Uppaal tool exist such as Uppal TIGA[8] , Uppal TRON[9] and ECDAR[10].

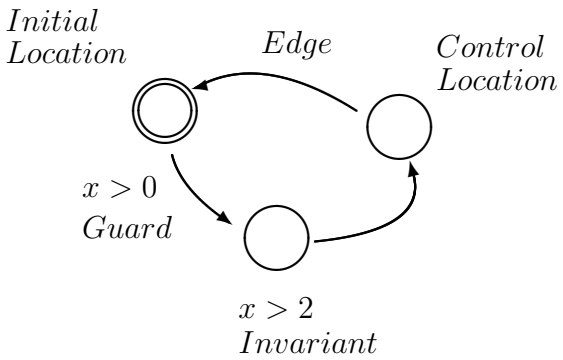


Figure 1.2: A timed automaton with a guard and an invariant

Timed automata are versions of finite state machines equipped with real-values clocks used for modeling real time systems. Invariants are placed on locations and boolean guards on edges (Figure 1.2). A state of computation of a timed automaton consists of a pair (l, v) where l is the control location the automaton is in ($l \in L$ where L is the set of possible locations) and v is the valuation determined by the current clock values. The pair (l, v) is a legal state of the timed automaton only if the valuation v satisfies the invariant of location l. If there is an edge whose source location equals the current location l and whose guard is satisfied by the current valuation v, then we can follow that edge, thereby changing the current location to the target location of the edge [3]. The action transition

[6] http://www.it.uu.se/
[7] http://www.cs.aau.dk/en
[8] http://people.cs.aau.dk/ adavid/tiga/
[9] http://people.cs.aau.dk/ marius/tron/
[10] http://people.cs.aau.dk/adavid/ecdar/

thus taken changes the system state instantaneously and the clocks can be reset when such transitions are performed.

The Uppaal toolkit is capable of describing reactive systems using timed automata extended with features for concurrency, communication and process priority. For that purpose it has a modeling language and a verification engine and has been extended with bounded discrete variables that are part of the state. Updates on transitions may be presented as functions in a C like syntax. A state of the system is defined by the locations of all automata, the clock values, and the values of the discrete variables.

The structure of a model description is divided into global and local declarations, automata templates and system definitions. The modeling language is compiled into a format that gives rise to visual representation and simulation of system models (Figure 1.1).

Uppaal also has a query language for expressing requirement specifications in TCTL (timed computation tree logic) [6] used by the model checker in the verification phase.

1.1 General problem statement

While Uppaal is capable of presenting the behaviour of reactive systems in a graphical way, BSV lacks a graphical interface to visualize the state changes of the system under investigation.

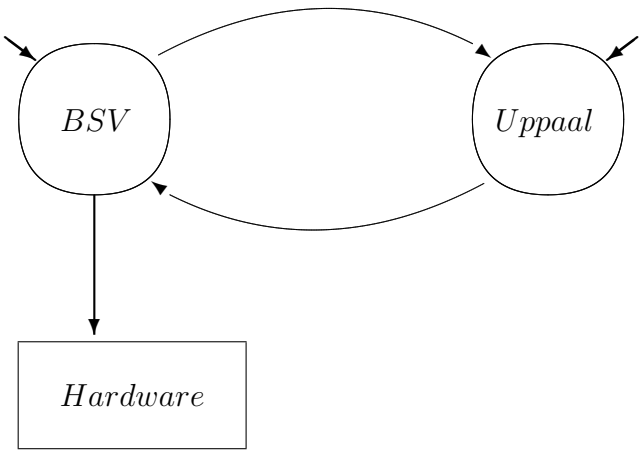


Figure 1.3: Preferable translations between two formalisms.

Our idea is to unify the advantages of both worlds so that a hardware specification written in BSV can be simulated and analyzed in Uppaal and conversely make a system description written and suitably investigated in Uppaal directly accessible for BSV to compile into hardware code. (See Figure 1.3. The in-going arrows indicate the two starting points in this approach.) This way the advantages of both types of formalisms can be utilized in order to obtain quality hardware.

Specifying hardware designs directly in Uppaal might require a new extension to the Uppaal tool in addition to those mentioned above.

1.2 Contributions

In this study we will provide means to translate from BSV to Uppaal. Doing so makes it possible to examine in detail the behaviour of a model specified in BSV within the Uppaal environment.

When BSV description language is compiled to hardware, all hierarchy is flattened into one top level module [17, p. 30] by default. This makes it possible for us to consider a BSV model as a collection of communicating automata in Uppaal.

Our approach will be to define a simplified version of BSV – which we refer to as Shallow BSV – and show that it can be translated into the Uppaal modeling language. As indicated in Figure 1.1 this will permit a full graphical examination of a system model in the Uppaal tool.

We will in this thesis construct a prototype of a front end compiler to translate from the shallow version of BSV (SBSV) into Uppaal (Figure 1.4). It accepts a hardware description written in SBSV and outputs a model description in XML format that can be analyzed in the Uppaal tool.

Our simplified language must also be compilable to BSV so that it can ultimately be compiled into executable hardware. Similarity in notation should make translation between BSV and SBSV fairly easy (dashed arrows).

Only a subset of programs written in BSV are synthesizable [17, p. 28] and we will limit our research to programs that can be directly compiled into hardware code. Our focus will be on the executable specifications which we will relate to similar constructs in the Uppaal language.

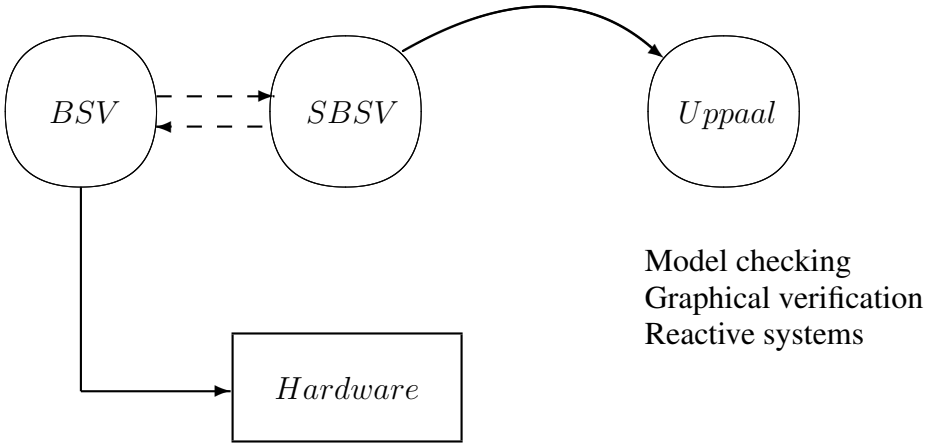


Figure 1.4: Simplified version of BSV translated into Uppaal

1.3 Structure of the thesis

The structure of the thesis is as follows: In the first two sections we introduce the two languages BSV and Uppaal in some detail. In Chapter 3 we build on the discussion in the previous section and provide an abstract grammar for the simplified version of BSV. Based on the grammar, Chapter 4 contains a description of a translator for SBSV and some examples of translations into Uppaal are shown. In Section 5 and 6 we finish off with conclusions, related and future work.

Chapter 2

Background

In the following we will give a brief introduction to the two languages considered in this thesis. The discussion will be informal with emphasis on the constructs that are of importance in our work.

2.1 Bluespec SystemVerilog

Bluespec SystemVerilog, or BSV, is a high level hardware description language where the atomicity of parallel rules together with precise handling of side effects allow for reasoning about a state of a model. BSV uses a discrete time model and state changes occur once every clock cycle.

BSV is object oriented where the objects, called modules, are the central constructs of the language and will be compiled into hardware components. Modules are defined using module definitions that correspond to class declarations in object oriented languages. The concrete modules are then obtained as instances of this general description. Modules form hierarchy by aggregation when a module description involves instances of other module descriptions.

The concept of modularity, where an isolated functionality has been defined in a module that has a well defined interface makes programming a much easier task. Modules are referenced and interacted through interface methods. This idea of higher level programming and modularity creates a framework for the programmer where (s)he does not need to know the details on how the underlying hardware works.

The aim of BSV is to enable design by refinement from executable specifications to final implementation. The language is described to be a high-level synthesizible specification language, which means that specifications can be directly converted into hardware.

2.1.1 Modules

BSV modules consist of declaration of variables, interfaces, subclasses and methods and of rules. A module definition corresponds to a class description in object oriented languages such as Smalltalk [12] or Java [13] and its instances, the modules correspond to the objects in these languages.

In the simple BSV program listed below the module *mkAdder* (line 17) is a submodule of the top module *mkTb*, that represents the starting point of the program execution.

Listing 2.1: A BSV program

```
1  package Tb;
2  (* synthesize *)
3  module mkTb (Empty);
4        Adder_ifc ifc <- mkAdder;
5
6        rule add;
7              $display (ifc.sum (10, 15, 17));
8              $finish (0);
9        endrule
10 endmodule
11
12 interface Adder_ifc;
13       method int sum (int x, int y, int z);
14 endinterface
15
16 (* synthesize *)
17 module mkAdder (Adder_ifc);
18       method int sum (int x, int y, int z);
19             return x + y + z;
20       endmethod
21 endmodule
22 endpackage
```

The purpose of the encapsulating *package* is only to define a name space and tell the compiler where to find the code, but does not correlate with the hardware structure.

Modules are instantiated and bound to a variable whereafter the methods implemented are accessible within the scope of the variable (line 4). A module hierarchy is created when a module is instantiated inside another, and the new object becomes a submodule. A statement of the form:

Interface type identifier <– module name;

invokes a module instantiation that returns a value of the interface type. It is then bound to the identifier. In the above example the variable *ifc* becomes a handle to the module *mkAdder*:

```
1       Adder_ifc ifc <- mkAdder;
```

Because a new instance different from earlier instantiations is created, this kind of a statement has side effects and is called side-effect initialization. After the initialization the method *sum* (line 18) implemented by the module can be referenced by qualifying the variable name as *ifc.sum()* (line 7) and communication between the two modules is established.

When the program is executed its only rule *add* is fired (once only due to the system function *$finish(0)* which halts the program) invoking the interface method that returns the integer 42 as a result.

Module definitions do not support inheritance although instances can be initialized with parameters when they are created.

2.1.2 Interfaces

An interface defines the communication of a module to its surroundings and other modules. A declaration of an interface contains a number of methods that describe the input and output of a module containing an instance of the interface. Methods can also be said to be grouped together to form an interface for a module. An example of an interface with two methods *receive* and *send* is:

```
1    interface Pipe_ifc;
2          method int receive ();
3          method Action send (int a);
4    endinterface
```

A module that accepts an interface as a parameter and implements its methods is said to *provide* the interface. The module *mkPipe* below provides the interface *Pipe_ifc* and must therefore implement the two methods:

```
1  module mkPipe (Pipe_ifc);
2     method Action send (int a);
3          x1 <= a;
4     endmethod
5
6     method int receive ();
7          return x4;
8     endmethod
9  enmodule
```

Several modules can provide the same interface without declaring the interface all over again – a well known paradigm used in high level object oriented languages. The non-blocking assignment notation (line 3) is a register write action[1, p. 52].

2.1.3 Methods

In the program listing 2.1 we saw how a top module communicated with a submodule through an interface method. In BSV there are three types of methods;

Value methods Return values from registers in a module.

Action methods Change the values of registers in a module (update of state elements).

ActionValue methods A combination of action- and value methods.

Methods may have an *implicit condition* as in the example below where *f2* can only be invoked when *x* is even:

```
1   method int f2 () if (is_even(x));
2      return x;
3   endmethod
```

The implicit condition may be a boolean expression or it may be a pattern-match, but cannot include updating of state elements. The method is executed only when the implicit condition is true or matches the pattern.

A method *m1* can be invoked from another method *m2*, but every method is ultimately invoked from a rule.

2.1.4 Rules

Rules are used to describe all behavior of a system (how state evolves over time) and are made up of two components;

Rule condition A boolean expression which determines if the rule body is allowed to fire[1].

Rule body A set of actions which describe the state updates that occur when the rules fire.

Rules in BSV are guarded commands and each rule consists of a boolean guard and a statement. They are the parts of the modules that are activated when a module is executed as will be explained below.

As an example we consider a rule where the values of two registers x and y are counted up and swapped. It has the following syntax:

```
1   rule r1 (x < y);
2      x <= y + 1;
3      y <= x + 1;
4   endrule
```

[1] The terms *fire* and *execute* are used interchangeably.

The actions in the rule body occur simultaneously and instantaneously and result in a changed state as register values are updated. The rule can fire at each clock cycle as long as x is less than y in which case the boolean guard evaluates to true.

The rule is an atomic unit in the sense that no intermediate values are recorded, but only the final values that are obtained at the end of their execution. All reading of variable values during the clock cycle refer to values *before* the execution of the rule. No update of variables takes place until at the end of the execution. One consequence of this is that the order of statements inside a rule is of no importance – as opposed to the common practice in imperative languages [4].

A program execution is event driven in the sense that at each clock cycle every rule in each module is a candidate to be fired. The boolean guards together with the execution order decided by a *Scheduler* determine whether a clock event causes a rule to fire or not.

The atomicity of rules allows for concurrency in BSV. Rules are allowed to be executed simultaneously as long as they do not update the same stateful variables and their guards are satisfied at the beginning of the clock cycle.

At the beginning of a clock cycle, several (maximal) sets of rules might be fired at the same time. When the hardware description is compiled, the schedule decides which set of rules fires in that clock cycle. Two different schedulers can result in different performance of the hardware even though they are compiling the same source code.

Although we have limited control over how the scheduler works, special attributes can be inserted into programs to overrule decisions made by the compiler. An example where we give rule *r1* precedence over rule *r2* is:

```
1   (* descending_urgency = ... ,r1, r2 ... *)
```

In this case we tell the compiler that *r1* is more urgent than *r2*.

As the execution model of BSV is clock cycle driven rather than sequential one has to indicate explicitly if a program is to halt. This is seldom of concern in reactive programs [3] as they usually have no final state in sight. For a single execution of a rule we have an example of such a statement in code listing 2.1 (line 8). Otherwise a dedicated rule is required to serve that purpose. An example of a rule that halts a program execution is the following:

```
1       rule done (x >= 30);
2             $finish (0);
3       endrule
```

The guard enables the rule when x reaches a certain limit in which case execution is stopped by the system function. No global variables are changed and the state remains the same.

The idea of using rules in BSV dramatically simplifies concurrent hardware design.

2.2 Uppaal

Uppaal is an environment toolkit for modeling, validating and verifying real time systems. A model is described in the Uppaal modeling language and compiled and run by a verification engine or server (Figure 2.1). A client with a graphical interface, implemented in Java, is used for observing simulations of a system and for verification by a model checker. The client is divided into editor, simulator and a verifier.

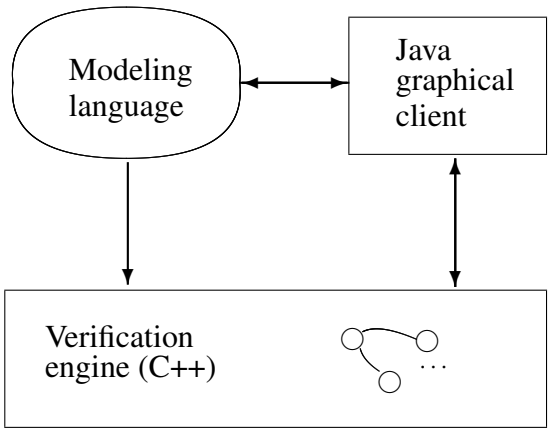


Figure 2.1: Components of the Uppaal environment

A model description in Uppaal is a collection of one or more timed finite state machines – or automata – that communicate through channels by using synchronization similar to the kind used in CCS [9]. The communication can be either rendezvous or broadcast synchronization.

When a model is designed in the client, two files are created, one containing the model description and the other holds the correctness specifications expressed in temporal logic.

Models can also be constructed by editing the file containing the model description without the aid of the client which is the proposed approach in this study (Section 4.1).

Uppaal uses a dense time model where clock values are represented by non-negative rational numbers. As the semantics of BSV is driven by clock cycles, and is therefore based on discrete time, we only use the untimed version of Uppaal in our modeling. A clock cycle in BSV, or even a firing of a single rule, may correspond to several transitions in Uppaal.

Uppaal consists of three main parts: a description language, a simulator and a modelchecker.

2.2.1 Description language

In the Uppaal language we have four predefined types *int*, *bool*, *clock*, and *chan*. Array and record types can then be defined over these and other types. Values of global variables determine the state of a system at any given time.

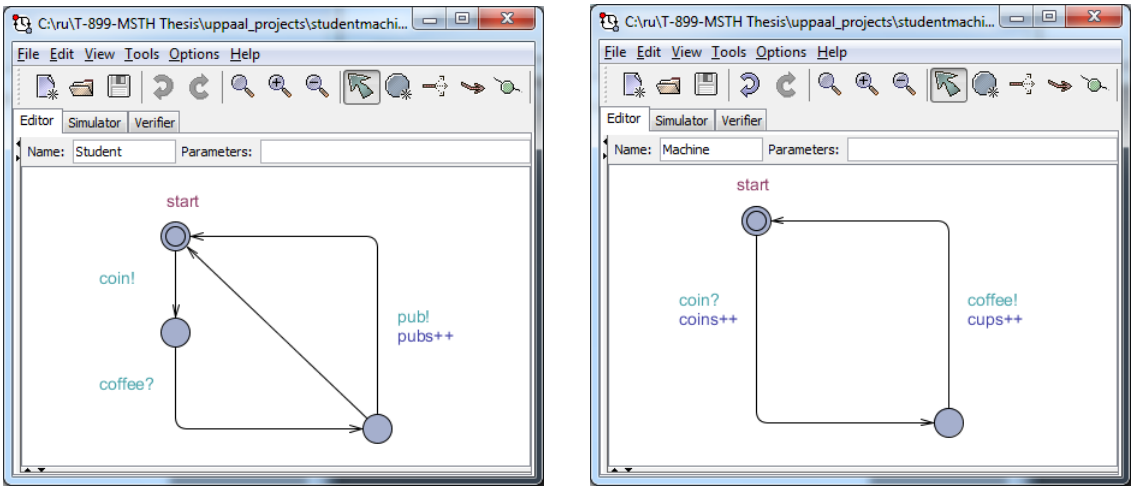
The structure of a model description is divided into global and local declarations, automata templates and system definitions as shown in the code listing excerpt below:

```
<declaration>
      int r = 0;
</declaration>
<template>
      <name x=5 y=5>Initializer</name>
      <declaration />
      <location id=id8 x=-496 y=-640>
            <name x=-504 y=-616>end</name>
      </location>
      <location id=id9 x=-608 y=-640>
            <name x=-616 y=-616>start</name>
            <committed/>
      </location>
      <init ref=id9/>
      <transition>
```

```
16              <source ref=id9/>
17              <target ref=id8/>
18              <label kind=assignment x=-576 y=-664>r = 23</g>
19      </transition>
20  </template>
21  <system>
22      initializer = Initializer();
23      system initializer;
24  </system>
```

The system contains a single automaton with two locations and an edge (transition). When the edge is taken, a constant is assigned to the global variable *r* (line 18) thus changing the state of the system. Instead of a simple assignment a function can also be associated with the transition, in which case the function is defined locally to the encapsulating template.

Another example[2] showing two processes communicating on channels *coin* and *coffee* is shown in Figure 2.2.



(a) Student process (b) Cofee machine process

Figure 2.2: Communicating automata in Uppaal

Apart from global variables that hold the state (not shown), channels for binary synchronization are declared in the Uppaal language:

[2] Based on [3] and http://www.comp.nus.edu.sg/ cs5270/Notes/chapt6a.pdf

```
1  <declaration>
2        chan coin, coffee, pub;
3        ...
4  </declaration>
```

When a synchronization is made between the two processes *student* and *machine* on the channel *coffee*, the value of the variable *cups* is counted up and the state of the system is advanced. If we were to pass a parameter between the two processes a special variable declared as *meta* would be used. A meta variable is reset at the end of each transition and is not part of the systems state.

The communication is expressed in the modeling language as follows (details omitted):

```
1  <template>
2        <name>Student</name>
3        <transition>
4              <label kind=synchronisation x=-284 y=-151>
5                    coin!
6              </label>
7        </transition>
8  <template>
9        <name>Machine</name>
10       <transition>
11             <label kind=synchronisation x=-236 y=-135>
12                   coin?
13             </label>
14             <label kind=assignment x=-236 y=-120>
15                   coins++
16             </label>
17       </transition>
18 </template>
```

The handshaking synchronization is expressed in lines 5 and 12 modeling the messaging between the two modules where the process *machine* accepts a coin from the process *student*. The full code listing can be found in Appendix B.1.

In the above example edges are taken in a non-deterministic manner. Another important type of location is a *committed* location. If a committed location is part of a system state the next transition must involve an outgoing edge from at least one committed location.

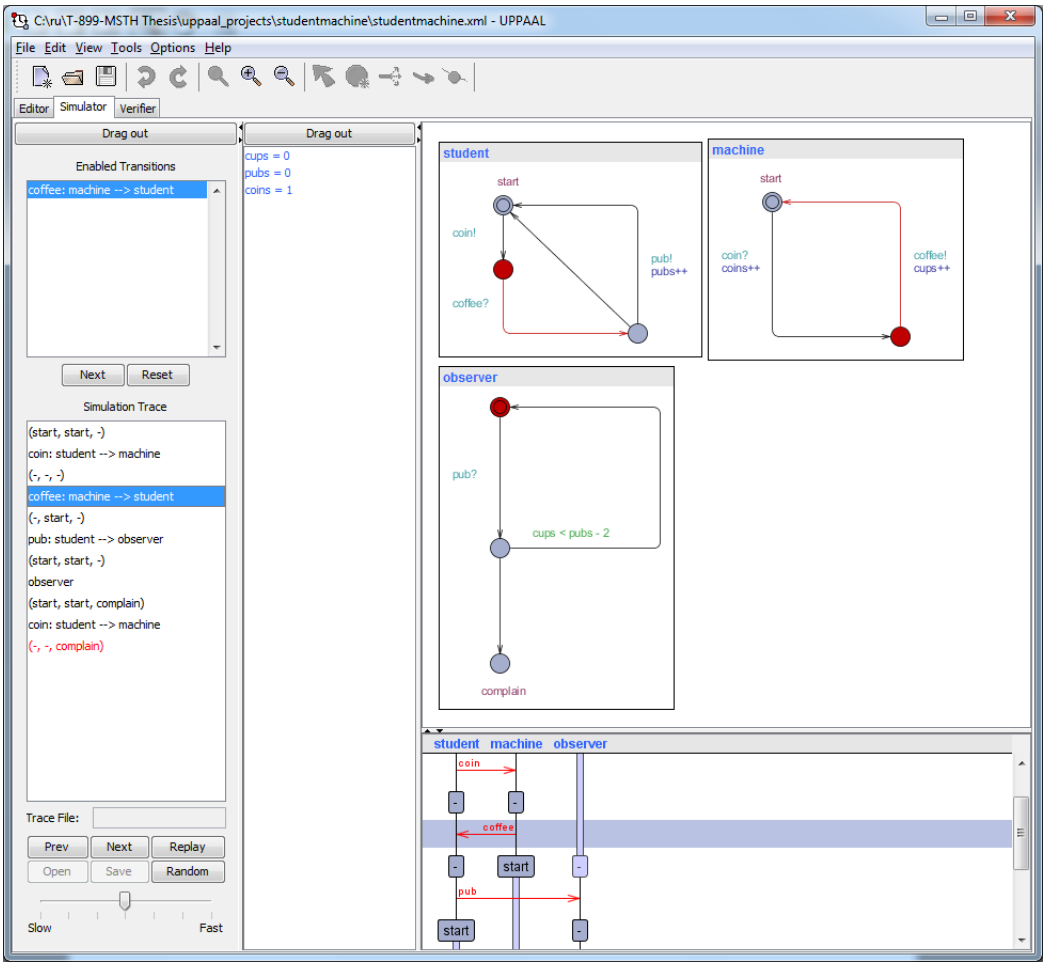


Figure 2.3: Simulation of a model in the Uppaal IDE

2.2.2 Simulator

Possible dynamic executions of a system in Uppaal can be examined by a validation tool during early modeling stages. The simulator provides means of fault detection prior to verification by the model checker which covers the exhaustive behavior of the system.

The integrated development environment (IDE)[3] shows the evolution of the system during execution from one state to the next (Figure 2.3). Various executions can be run in the simulator and some useful tests can be performed. Although this is a powerful way to assess a systems behaviour it is not adequate for larger systems where a modelchecker is needed.

2.2.3 Modelchecker

Uppaal also allows for model checking where the user can provide a correctness specification in temporal logic in the verifier and get true or false as a response. The modelchecker explores the state space of the system and checks invariant and reachability properties given in TCTL formulas.

By virtue of the modelchecker one is able to reason about dependability and reveal unwanted behaviour of a system such as potential deadlocks at design time rather than in the finished hardware.

[3] Current version is 4.1.13 as of this writing

Chapter 3

Translating BSV into Uppaal

As described earlier, the main aim of this study is to provide a translation of a BSV program to Uppaal so it can be thoroughly investigated before it is permanently turned into hardware. As we already pointed out in Section 1.2, an intermediate state of the BSV compiler involves flattening out the module hierarchy into a collection of modules that all are on the same level. This intermediate language is much closer to the spirits of a Uppaal system that consists of a collection of automata. Therefore, instead of translating a general BSV term directly into a Uppaal, we take advantage of this form and base our translation on it. Unfortunately we do not have access to the intermediate language the BSV compiler produces, so instead we define our own version which we refer to as Shallow BSV (SBSV). For the sake of convenience we do not follow exactly the standard BSV notation but our notation can be very easily translated into BSV (dashed lines in Figure 3.1).

3.1 Shallow BSV

A SBSV program can be considered to be a collection of modules and a set of global variables accessible from inside every module. Following the BSV trend, a SBSV program is represented as a package containing these components. Each module consists of its local temporary variables and rules. Unlike in general BSV, no methods are defined inside a module in SBSV and no submodules inside a module are allowed either. Furthermore no class definitions are allowed in SBSV and instead modules are defined directly one by one.

In standard BSV, registers contain values that decide the state of the system. The result of running the program is therefore to change the content of the registers. BSV also allows temporary variables that are used to store temporary values during computations within clock cycles but are reset after each cycle.

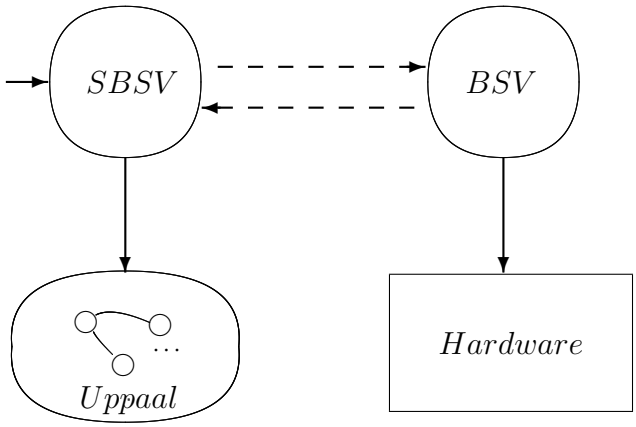


Figure 3.1: Shallow version of BSV translated to Uppaal for model checking

Based on these considerations, the variables that are declared in SBSV are of two types. On the one hand, we have temporary variables that only hold values over time in computations within the clock cycle. They are reset at the end of the clock cycle according to the syntax where they are declared, and do not count as part of the state of the system and can either be global or local to each of the modules of the system.

On the other hand, we have the stateful variables of type *register* that are updated during the clock cycle and the value they contain at the end of the cycle will be carried over to the following cycle. The states of the system are decided by the value these variables have at the beginning of each cycle. Which values they might have between the clock signals is irrelevant in this sense. These variables are always global and are declared at the top of the program. They can therefore be accessed by any of the modules of the system.

Like in standard BSV a rule consists of a boolean and a statement. As we mainly focus on the structure of the modules, for the sake of simplicity we assume that both the expressions and the statements that are allowed in SBSV are basically the same as in Uppaal that inherits these notions from C like languages. We also assume that all values

are either natural numbers or Booleans. Boolean expressions in SBSV do not have side effects.

As the maximal set of rules is chosen to fire in each clock cycle, independent of which module they belong to, the number of rules in a module is not relevant. Therefore SBSV allows only one rule in each module which considerably simplifies the translation process (Chapter 4).

3.2 The grammar for SBSV

To describe formally the grammar for SBSV, we use the standard BN notation like we did for the full language in [16, ch. 3 - 9]: Alternatives are separated by a vertical bar (|), items enclosed in square brackets ([ ]) are optional, and key words are given by bold face text. The variables are given by italic text. The following abstract grammar describes SBSV. As the syntax for statements and expressions are straight forward they are omitted:

package	::= **{** *decls moduleDef* **}**
moduleDef	::= **module** *varDecls rule* **endmodule** *moduleDef*
rule	::= **rule** *identifier* [*condition*] **;** *statement* **endrule**
decls	::= *varDecl regVarDecl*
varDecl	::= *intVarlDecl boolVarDecl*
intVarDecl	::= **int** identifier **=** numeral **;** *intVarDecl*
boolVarDecl	::= **bool** identifier **=** boolean **;** *boolVarDecl*
regVarDecl	::= **reg** identifier **;** *regVarDecl*

3.2.1 Code example

Program listing 3.1 shows an example of a syntactically valid SBSV program. The example contains a hardware description known as *rigid synchronous pipeline of registers*[1]. Two modules, *M1* and *M2* simultaneously update a set of state variables. Five registers

[1] Based on a similar example for BSV [1, p. 58].

are initialized and on each rule firing each register is incremented and shifted into the next register. As in standard BSV these shifts all happen simultaneously i.e. all the shifts are performed in one clock cycle.

Listing 3.1: A valid SBSV program

```
1  {
2     register x1 = 10;
3     register x2 = 20;
4     register x3 = 30;
5     register x4 = 40;
6     register x5 = 50;
7
8     module M1();
9       rule r (true);
10          x2 = x1 + 1;
11          x3 = x2 + 1;
12      endrule
13    endmodule
14
15    module M2();
16      rule r (true);
17          x4 = x3 + 1;
18          x5 = x4 + 1;
19      endrule
20    endmodule
21 }
```

Even though both modules are accessing the same register (*x3* in line 11 and 17) the SBSV semantics allows parallel execution of the two rules.

3.3 Informal semantics of SBSV

As SBSV is a sublanguage of BSV (modulo minor syntactic differences) it inherits of course its semantics too. However, to better understand the translation of the language into Uppaal, we will give an intuitive explanation of its meaning.

Like in BSV the semantics of SBSV is based on clock cycles. At the beginning of each cycle the registers and the temporary variables contain values that can be read by all the

modules according to their scope. The modules read the values from the registers before the clock cycle but cannot read from it during that cycle. The module can however write in the register at any time but that value may be overwritten by another module. The temporary variables are always reset at the beginning of each clock cycle but can, on the other hand, be written into and read from during a cycle. A clock cycle can be completed at any time indicating that either there are no more rules to perform or that the energy is used up in that cycle.

Parallelism in standard BSV takes place when more than one rule, within the same or different modules, is fired at the same time given that their guards are satisfied in the current state [16, ch. 6.2.2]. The BSV semantics requires that rules that fire in parallel do not write into the same registers. This approach carries mostly over to SBSV with the exception that SBSV, as explained above, allows more than one rule to write into the same register in the same cycle but only the last writing will be recorded at the end of the cycle. As no reading is allowed of the registers inside cycles, these two approaches are semantically equivalent although clearly one is more time and energy effective than the other. In this study we only focus on properties that have to do with possible states, we adopt the more simplistic approach and look at the other as a scheduling problem.

3.4 Translation of SBSV into Uppaal

In the above we have described a SBSV program as a collection of globally defined variables and modules each with its set of local temporary variables and rules. This leads to the following guidelines for the basic structure of translation of SBSV into a collection of interacting automata in Uppaal.

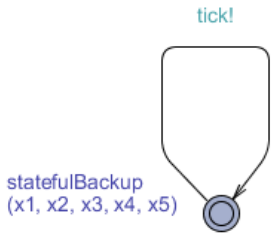
Registers are the only stateful elements of the SBSV system. As they are basic constructions of the language, each maps to a pair of global variables, one holding the current value of the register and the other is used for backing up the register value at the start of a clock cycle:

```
1   SBSV                          Uppaal
2
3   register x                    int x
4                                 int xStart
```

The global variable *x* represents the state of the register and carries its value from the end of one clock cycle to the beginning of the next one. At the beginning of each cycle, *xStart* is assigned the value of *x* and after that, all readings of the value of the register during the clock cycle are done from *xStart*.

The role of the global variable *xStart* is to store the initial value of the register *x* unchanged throughout the clock cycle. This secures that all the modules use the register values as they are at the beginning of the clock cycle, even if some of them write into some registers during that cycle. Therefore executing several rules or statements one after the other in the Uppaal clock cycle loop has the same effect as if they were executed is parallel.

The compiler begins by creating a **clock** automaton that coordinates the start of every new clock cycle. It has a single transition loop labelled with the backup statement *statefulBackup* and a send action *tick!* on a broadcast channel (Figure 3.2)(a). The interval between ticks corresponds to a clock cycle in BSV.



(a) A clock automaton

```
void statefulBackup(int v1, int v2, int v3,
                    int v4, int v5)
{
    x1Start = v1;
    x2Start = v2;
    x3Start = v3;
    x4Start = v4;
    x5Start = v5;
    i++;
}
```

(b) Backup of state variables

Figure 3.2: Implementation of a clock cycle in Uppaal

The role of the clock in the system is to coordinate the firing of rules and backing up the stateful values at the start of each clock cycle. The procedure *statefulBackup* accepts the state variables as parameters and assigns their values to the temporary *xStart* variables (Figure 3.2(b)).

A **module** translates into an automaton with one location and one looping transition. The loop is labelled with the translation of the rule of that module (the guard and the statement) and a *tick?* co-action for synchronizing with the clock. The loop might have some committed locations[2] to secure atomicity.

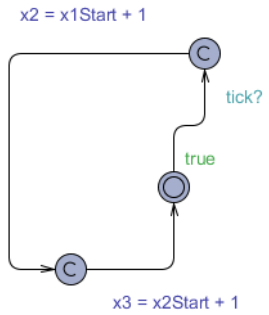
[2] A committed location cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations in the model (Section 2.2.1).

When **rules** are translated into a loop each occurance of x on the right hand side of an assignment is raplaced with *xStart* i.e. all readings from x become readings from *xStart*.

Following these guidelines and the semantics of SBSV, the module *M1* from program listing 3.1 translates as shown in Figure 3.3

```
module M1();
   rule r (true);
        x2 = x1 + 1;
        x3 = x2 + 1;
   endrule
endmodule
```

(a) SBSV module



(b) Uppaal automaton

Figure 3.3: Translation of a module into Uppaal

Summary

We can devide the translation from SBSV to Uppaal into the following steps:

1. An automaton representing a system clock is created. The clock contains a transition with one broadcasting channel used to synchronize with other modules.
2. For each state variable of type *register* two global variables are declared. One for holding the state value and the other, a temporary variable, that gets assigned the value of the former every time a synchronization is made with the clock – at the start of a clock cycle.
3. At last an automaton is outputted for every module in the system. The automaton contains transitions decorated both with the guards and the translated statements, where each occurance of x on the right hand side of an assignment is replaced by *xStart*, as well as a binary synchronization on the clock channel.

In Section 4.4 we run the example code (Listing 3.1) through the translator we provide for SBSV and show how the model can be simulated and model checked in the Uppaal IDE.

3.5 From Uppaal to BSV

Although it is not the aim of this thesis to show how Uppaal translates into BSV we will give an example of how that translation might work. A simplified version of the Jobshop example from [22] is shown in Figure 3.4. The model consists of two processes, a Jobber and a Hammer. The jobber makes a non deterministic choice between doing easy and hard work. For the hard work a hammer is needed, so by issuing a send action *get_hammer!* the jobber can continue only if the hammer performs the corresponding co-action *get_hammer?*. This has the effect that the hammer is taken and temporarily unavailable to other processes, that is the hammer is a shared resource. A similar synchronization is made when a task is finished. An upper limit is on the number of tasks performed and finally a dead state is reached.

As in the previous example we relate guarded transitions equipped with handshake synchronization in Uppaal to atomic guarded rules that implement an interface. The two automata *jobber* and *hammer* turn into two modules that communicate through interface methods.

A module in BSV provides an interface by implementing its methods. By doing so modules can exchange information and interact with each other. One way of modeling the jobshop scenario in BSV is by letting a hammer module implement an interface *Hammer_ifc* of the following form:

```
1   interface Hammer_ifc;
2       method Action getHammer();
3       method Action putHammer();
4       method Bool isFree();
5   endinterface
```

The module *mkHammer* (Listing 3.2) accepts the interface as a parameter (line 1) and has a private boolean variable *hammerIsTaken* that the two action methods[3] use to keep track of the hammers status. The value is local to the hammer and only accessible from the external environment through the interface methods *getHammer* (line 16) and *putHammer* (line 20) implemented by the hammers module.

[3] Action methods affect the state of the system whereas value methods like *isFree* do not.

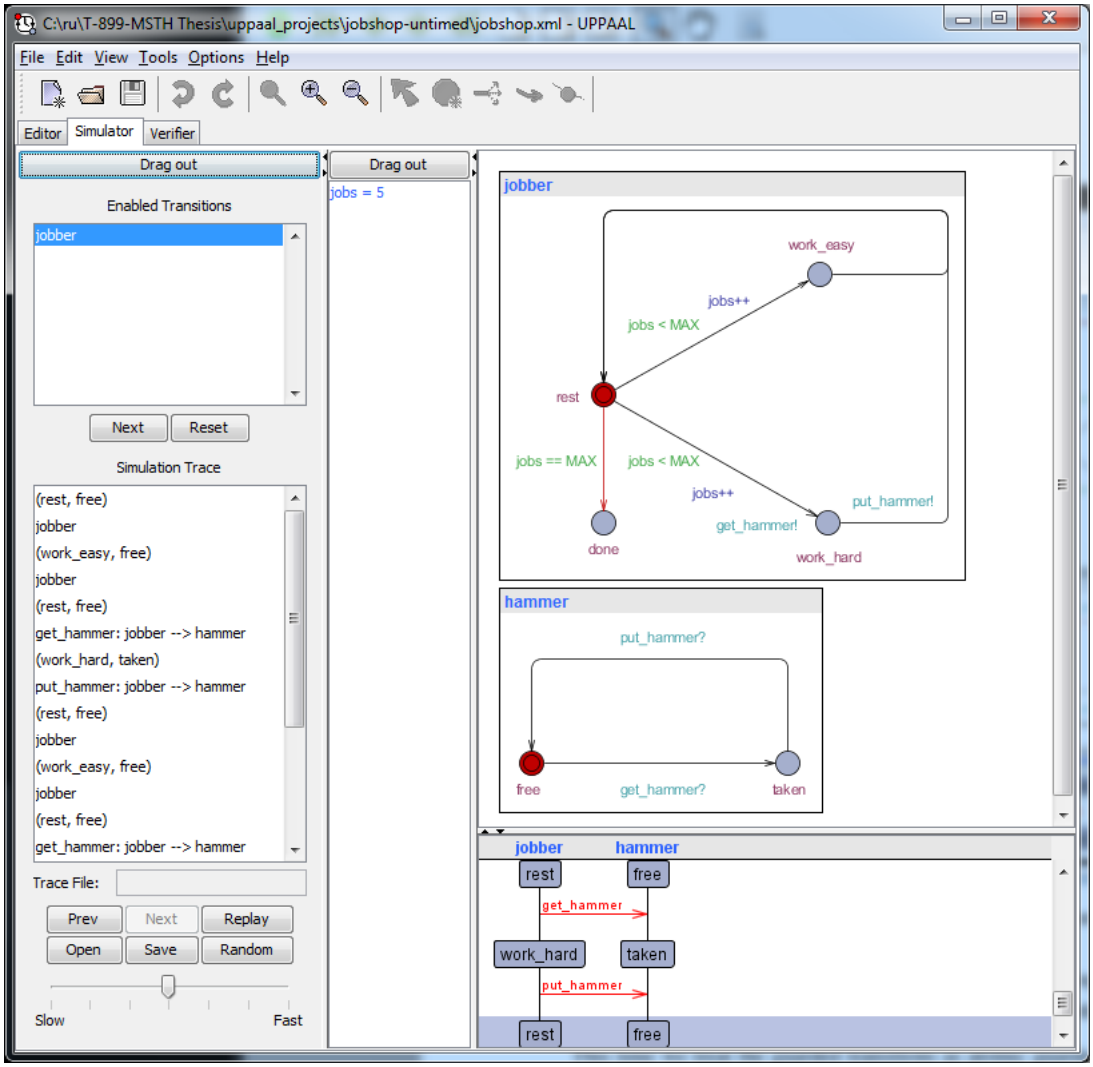


Figure 3.4: A simple Jobshop in the Uppaal environment

We give the testbench topmodule *mkJobshopTb* a handle to the hammers interface by instantiating:

```
1       Hammer_ifc hammer <- mkHammer (1);
```

By qualifying the variable names like in the rule condition and rule body below we have access to the methods:

```
1   rule work_hard if (hammer.isFree());
2     hammer.getHammer();
3   endrule
```

Listing 3.2: A module that provides an interface

```
1  module mkHammer #(parameter int init_val) (Hammer_ifc);
2    Reg#(Bool) hammerIsTaken <- mkReg (False);
3
4    function Action takeHammer();
5      return action
6        hammerIsTaken <= True;
7      endaction;
8    endfunction
9
10   function Action placeHammer();
11     return action
12       hammerIsTaken <= False;
13     endaction;
14   endfunction
15
16   method Action getHammer() if (hammerIsTaken == False);
17     takeHammer();
18   endmethod
19
20   method Action putHammer() if (hammerIsTaken == True);
21     placeHammer();
22   endmethod
23
24   method Bool isFree();
25     return hammerIsTaken == False;
26   endmethod
27 endmodule
```

The interesting thing here is that the communication is not through a global variable and does not affect the system state. This resembles the actions of the binary channel synchronization in Uppaal.

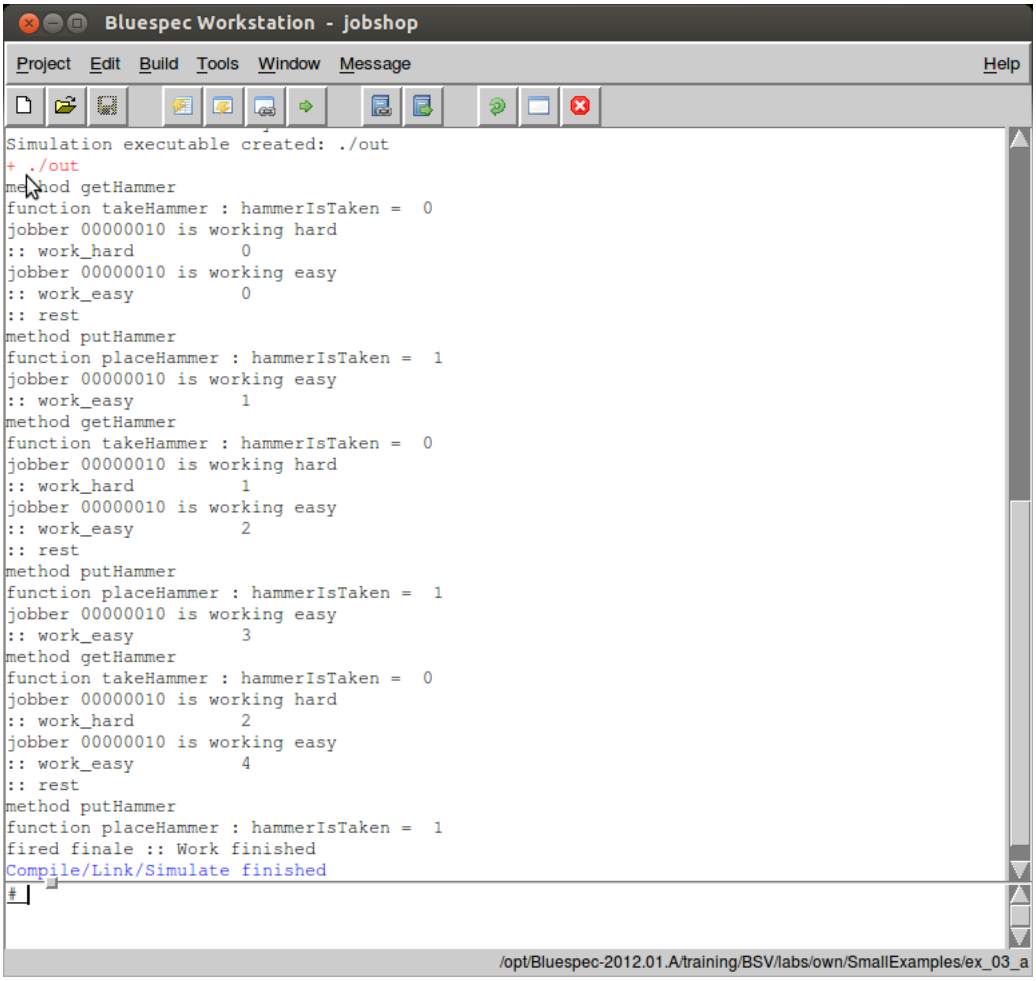


Figure 3.5: A BSV simulation of the simplified Jobshop

A run of the Jobshop example modeled in BSV is shown in Figure 3.5. The behaviour of the two models is so similar that the trace in one is reproducible in the other. Compare with the Uppaal simulation in Figure 3.4.

Chapter 4

Translator

We will now proceed to building a translator for SBSV that outputs a file containing Uppaal modeling language code. The file can then be opened in the Uppaal IDE for further examination and verification of the system model.

4.1 Front end of the compiler

In the previous chapter we developed a syntax for SBSV described by the grammar in Section 3.2. In the following we will build a translator that accepts a valid SBSV program and outputs a well formed Uppaal modeling language description.

The translator is written in Java and is based on the front end compiler described in *Compilers: Principles, Techniques, and Tools* [14, ch. 2.5 -2.8, 6.6] where syntax for a simple language is described and constructs in the language are modeled as Java classes. The classes output the appropriate intermediate language code for each construct. We find this approach attractive in this context because of the object oriented architecture of the compiler and will extend the code to suit our purposes.

A compiler that maps a source code to a semantically equivalent target program — usually for machine level — does so in two steps. The *analysis* part collects information about the source program by lexical analysis and outputs an intermediate representation to the synthesis part. In the *synthesis* part it is further compiled or interpreted and optimized for different machine platforms.

Figure 4.1 is taken from [14] and shows the phases of a compiler. We have marked what is often refereed to as the *back end* of a compiler with dashed lines to indicate that it is

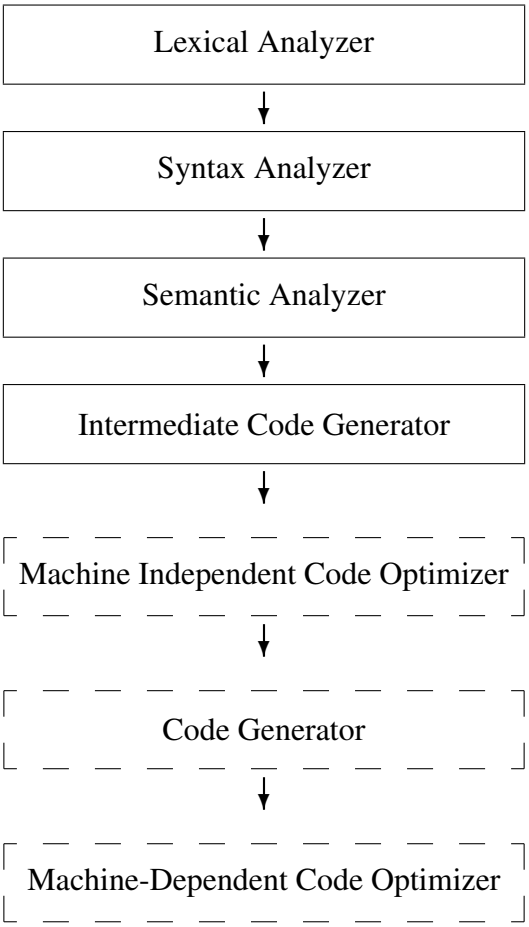


Figure 4.1: The phases of a compiler

outside the scope of this discussion. Our translation only passes through the front end for the reason that we treat the intermediate language as our desired target language.

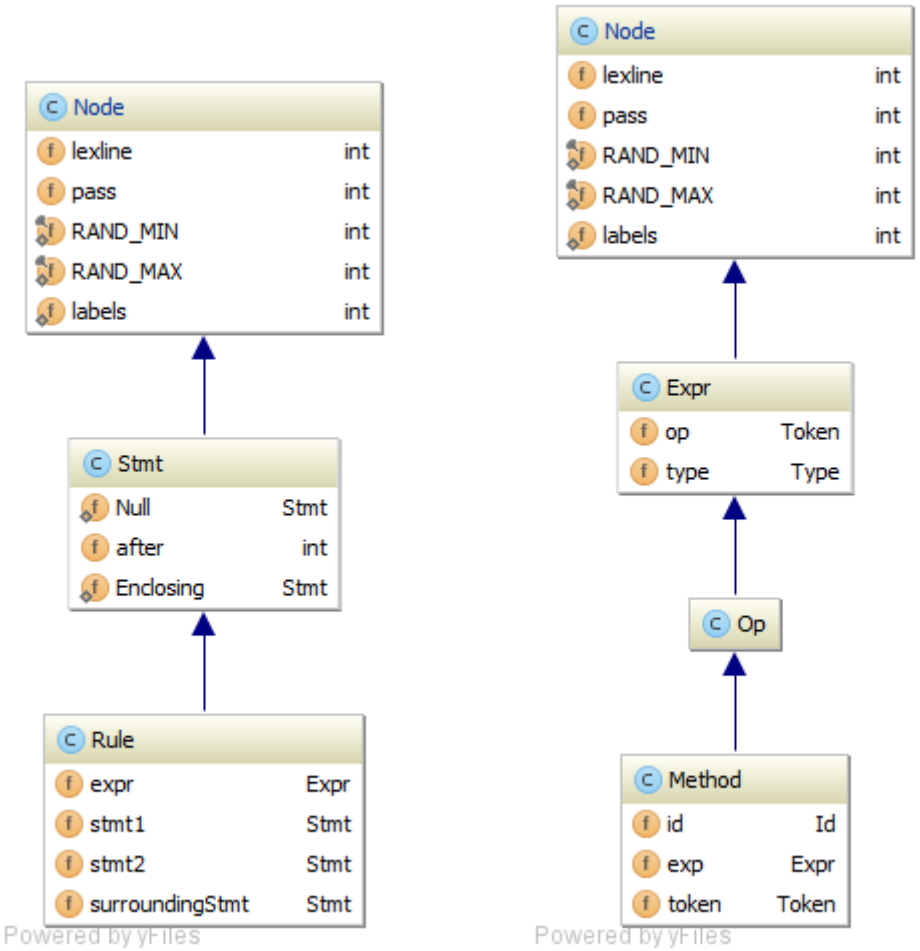
4.2 Program fragments attached to productions

In syntax directed translation, program fragments are attached to productions in a grammar. Appropriate productions for the various language constructs, such as expressions, statements and in our case modules and rules are the result of the parsing process.

An example of a production derived from the grammar in Section 3.2 is:

rule -> **rule** *r* (*expression*) **;** *statement* **endrule**

After the lexical analysis, the source program consist of a syntax tree with constructs or nodes implemented as objects. For instance, the lexical element **rule** is represented in the compiler by the class *Rule* shown in Figure 4.2(a).



(a) Inheritance hierarchy for class Rule (b) Method is a subclass of expression

Figure 4.2: Inheritance relations

To briefly outline how the translator generates Uppaal description language, we will in the next few sections describe how the code for the SBSV rule is generated.

4.3 Intermediate code generation

The format of the Uppaal description language is XML[1]. Locations and transitions are represented as nodes in the XML Document Object Model (DOM) class, that is an in-

[1] See e.g. Listing B.1 in Appendix B.1

memory tree-like representation of the model description. The DOM allows us programmatically to modify the tree and gradually build up the modeling language code in a non-sequential way.

Our translator is written in Java, so the Java Document Object Model (JDom) package [18] seems to be a natural choice for manipulating with the DOM tree in order to match the Uppaal syntax.

4.3.1 Outputting a rule

An automaton for a module is generated by a method *emitModule* in the Module class, that subsequently contains a call to a method *emitRule* from the *Rule* class to output the transitions for the rules in the module.

Listing 4.1: Excerpts of the Rule class

```
1
2  public class Rule extends Stmt
3  {
4     Expr expr;
5     Stmt stmt1, ...;
6     Stmt surroundingStmt;
7
8     public void setSurroundingStmt(Stmt surroundingStmt) {
9           this.surroundingStmt = surroundingStmt;
10    }
11       ...
12    @Override
13    public void gen() {
14              ...
15              emitRule(expr, stmt, surroundingStmt);
16              ...
17       }
18 }
```

The class Rule (Listing 4.1) has a variable *surroundingStmt* that is a handle to the module to which the rule belongs. The call to the method *emitRule* (line 15) accepts the surrounding module together with an expression and a statement as input parameters. The method outputs the Uppaal code for the SBSV rule.

Listing 4.2: The method emitRule in the Node class

```
1  public class Node
2  { ...
3
4  public void emitRule(Expr expr, Stmt stmt, Stmt surroundingStmt)
5  {
6        Module module = (Module) surroundingStmt;
7        Element orphan = new Element(_template_);
8
9        for (Element e : UppaalJDom.getInstance()
10                                    .root
11                                    .getChildren(_template_)){
12             if ( e.hasAttributes() && e.getAttributeValue(_name_)
13                                    .equals(module.getId()
14                                    .toString())){
15                   orphan = e; // find the location in the dom
16             }
17       }
18
19       if (expr instanceof Method){
20             Method method = (Method) expr;
21             orphan.addContent(new Element(_transition_)
22                   ...
23                   .addContent(new Element(_label_)
24                         .setText(expr.toString())
25                         .setAttribute(_kind_, _guard_)...)
26                   .addContent(new Element(_label_)
27                         .setText(method.id.toString()
28                         .toLowerCase()
29                         + expr.toString() + _!_)
30                         .setAttribute(_kind_, _synchronisation_))
31                   .addContent(new Element(_transition_)
32       ...
33       }
34 }
```

The placement of the transition in the DOM tree is determined (lines 9 to 17) and the appropriate tag for the transition is created and attatched (line 21).

4.3.2 Code generation for module communication

When a SBSV program contains a module that communicates with other modules, it is translated to an automaton that contains binary synchronization in Uppaal. The class Method (Figure 4.2) has a method *emitSynchronisationTransition* that outputs a transition with a channel action for that purpose [23].

4.4 Translation examples

Parallel composition

In BSV, values of two registers can be swapped in a single clock cycle:

```
1  module mkTb (Empty);
2
3        Reg#(int) x1 <- mkReg (7);
4        Reg#(int) x2 <- mkReg (14);
5
6        rule r1;
7           x1 <= x2 + 1;
8           x2 <= x1 + 1;
9        endrule
10
11 endmodule
```

The code initializes two registers *x1* and *x2*. The values are read before the rule-firing instant, and write of the new (+1) values takes place after the rule-firing instant. After the rule executes, *x1* has 1+ the old value of *x2*, and x2 has 1+ the old value of x1, effectively a swap of x1 and x2 along with some incrementing.

Similarly parallel composition, where two or more actions are composed in parallel to form a more complex action, can be achieved in SBSV:

```
1  {
2        register x1 = 7;
3        register x2 = 14;
4
5        module M1();
6           rule r (true);
7               x1 = x2 + 1;
8               x2 = x1 + 1;
9           endrule
10       endmodule
11 }
```

When translated, a clock and an automaton for the module *M1* are created. Simulation in Uppaal of a single clock cycle is shown in Figure 4.3[2].

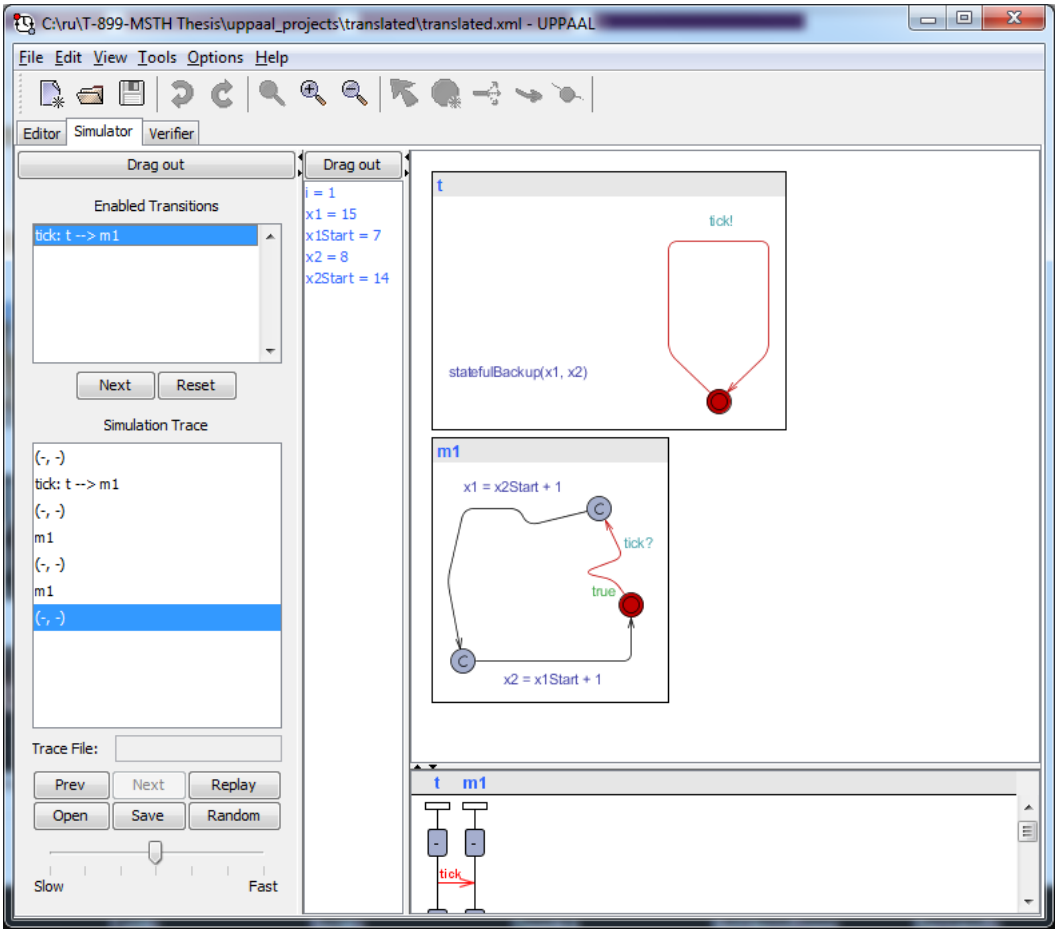


Figure 4.3: Register swap in the Uppaal IDE

[2] The coordinates of edges and locations are randomly generated explaining the fuzzy edges.

The variable i is local to the clock and keeps track of the number of clock cycles. From the figure we see that the initial values of the two registers (7, 14) have been incremented and swapped (15, 8) with the counter value still being equal to 1.

Greatest common divisor

The parallel version of Euclid's algorithm[3] can in BSV syntax be written as two processes with guards *g1* and *g2* as follows:

Action swap: g1 = ((x > y) && (y != 0))

x <= y; y <= x;

Action diff: g2 = ((x <= y) && (y!= 0))

y <= y - x;

If we relax the condition in the guard *g2* to secure that the values of *x* and *y* end up being equal, the algorithm takes the following form in SBSV:

```
1  {
2        register x1 = 18;
3        register x2 = 81;
4        register x0 = 0;
5
6        module M1();
7          rule swap ( x1 > x2 && x2 != x0 );
8              x1 = x2 + 0;
9              x2 = x1 + 0;
10         endrule
11       endmodule
12
```

[3] "The research, the outcome of which is reported in this article, was triggered by the observation that Euclid's Algorithm could also be regarded as synchronizing the two cyclic processes "do x:= x - y od" and "do y := y - x od" in such a way that the relation x > 0 and y > 0 would be kept invariantly true. It was only after this observation that we saw that the formal techniques we had already developed for the derivation of the synchronizing conditions that ensure the harmonious cooperation of (cyclic) sequencial processes, such as can be identified in the total activity of operating systems, could be transferred lock, stop and barrel to the development of sequential programs as shown in this article." – Dijkstra [7]

```
13        module M2();
14           rule diff (x1 < x2 && x2 != x0 );
15               x2 = x2 - x1;
16           endrule
17        endmodule
18  }
```

A translation gives the system shown in Figure 4.4[4]. After a few clock cycles the values of *x1* and *x2* is the greatest common divisor (9) for the two initial values (18 and 81).

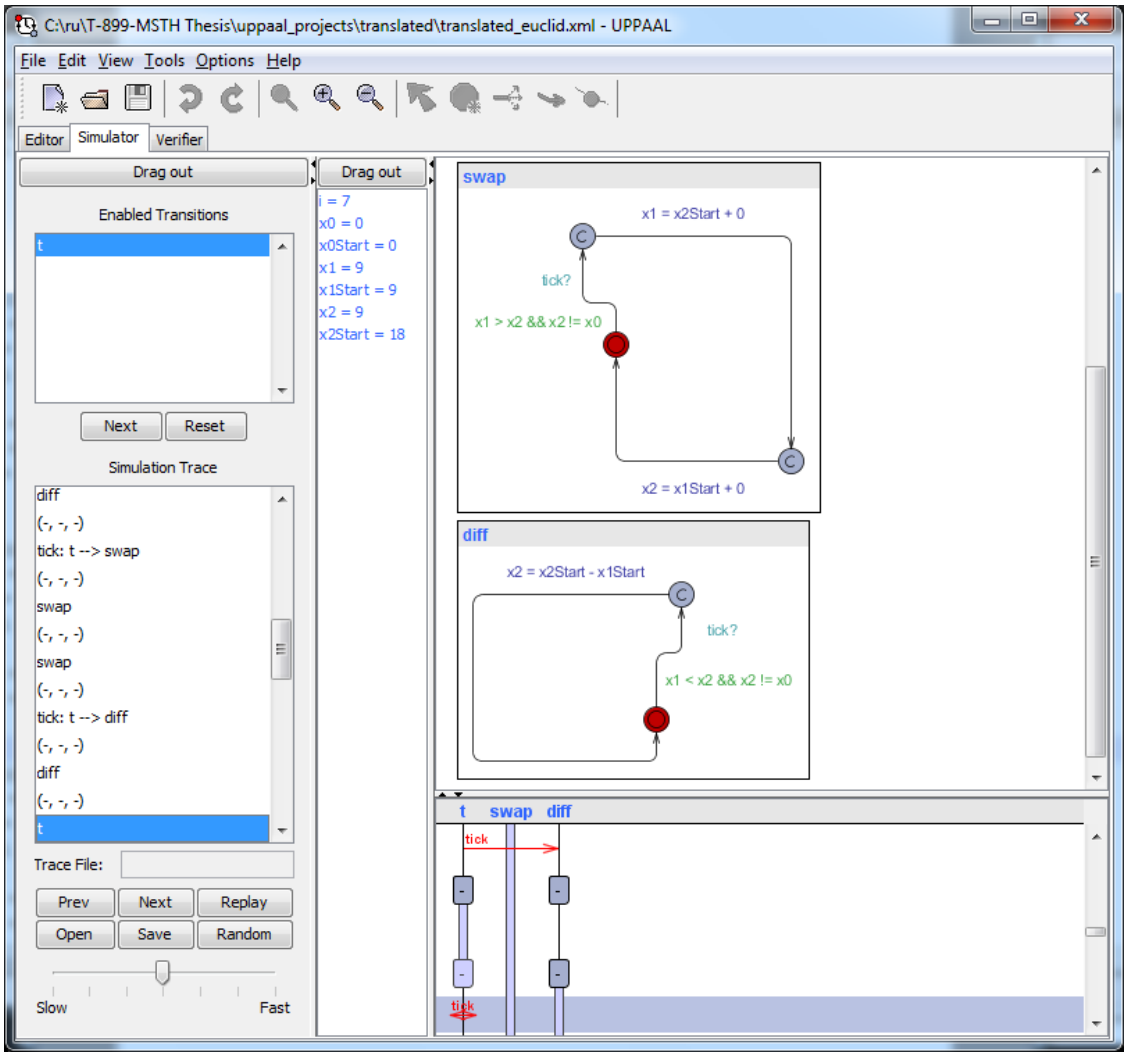


Figure 4.4: Greatest common divisor in Uppaal (clock not shown)

[4] Adding a zero in rule *swap* is a minor compiler workaround.

Rigid pipeline

The example introduced in Section 3.2.1 is a rigid synchronous pipeline of registers. The code in the editor pane of the translator[5] (Figure 4.5) is from code listing 3.1.

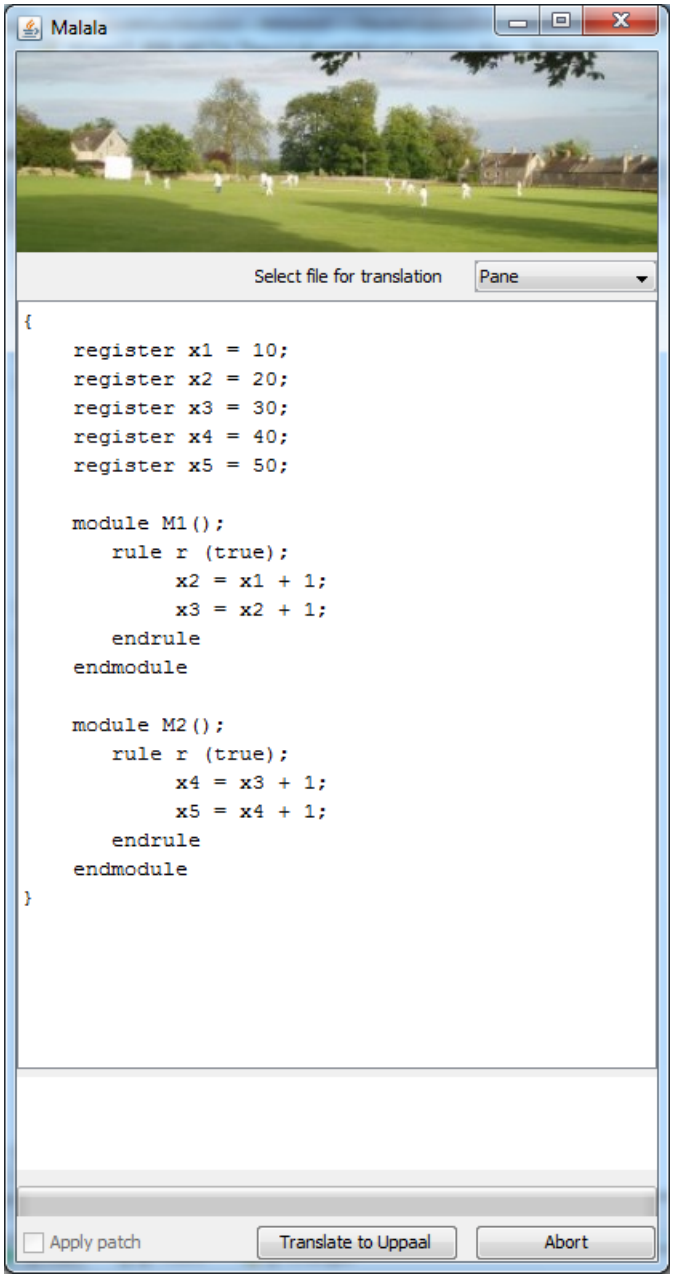


Figure 4.5: The user interface of the Malala application

The values of the registers $x1$, $x2$, $x3$ and $x4$ are simultaneously shifted in one clock cycle. We could have used one rule that translates into a single transition, but that would

[5] Named after the education activist Malala Yousafzai

require a scheduler of the kind discussed in Section B.2. Instead we implement a rigid synchronous pipeline by having one rule per module.

By pressing the button *Translate to Uppaal* a file in XML format is written. Figure 4.6 shows the system in the Uppaal IDE.

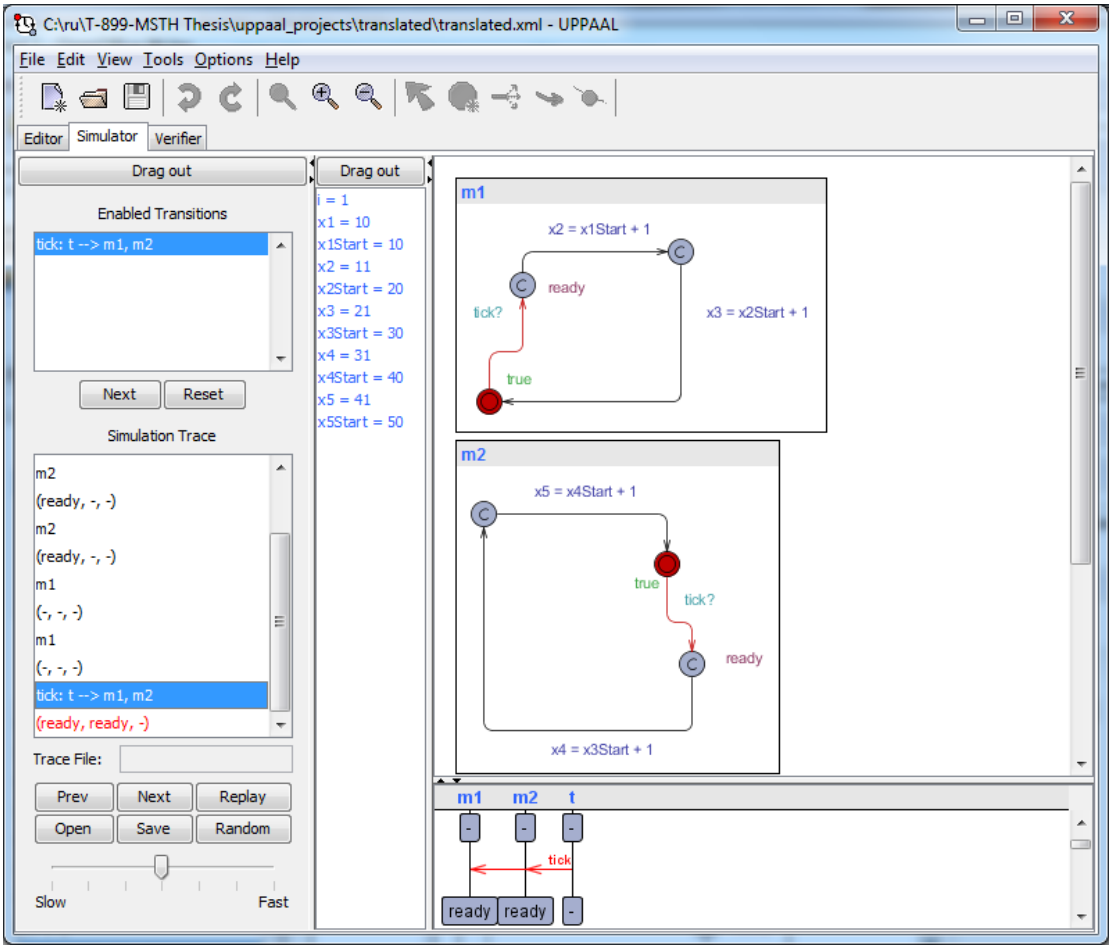


Figure 4.6: A simulation of a rigid pipeline (clock not shown)

The initial values of the registers is (10, 20, 30, 40, 50). In the simulation the state variables have been incremented and shifted, giving the state (10, 11, 21, 31, 41). Notice that the value of the counter i, that counts the number of clock synchronizations, is 1 at this stage. The initial synchronization with the clock is also shown in the lower half of the figure.

Model checking

The main benefit of translating from a hardware description language like SBSV, to a framework with a verifier such as Uppaal, is that we can check various temporal properties of a system.

Refering to the GCD example (Figure 4.4) the first property below states that invariantly the two variables *x1* and *x2* are greater than zero.

```
1 A[] x1 > 0 && x2 > 0
2 E<> x1 == x2
```

When the property is checked by the verification engine of Uppaal, it gives *The property is satisfied* as a result or *true*. According to the program listing, the greatest common divisor is found when the variables *x1* and *x2* are equal. The second property checks if there exists a computation path where that is the case and it also evaluates to *true*.

By manually naming locations, after the translation, one can check if a process reaches certain states. For the rigid pipeline example (Figure 4.6) we have named two locations *ready*. A property stating that the there exists a computation path where both modules are in location *ready*, and a relationship between variables holds, is shown below:

```
1 E<> m1.ready && m2.ready && (x4 < x4Start || x4 > x4Start)
```

The verifier returns *false* in this case and we can conclude that just after the synchronization with the clock, it is impossible that the value of the state variable *x4* differs from the temporary variable *x4Start*. The same goes for the other variables, indicating that the clock automaton works as intended.

The need for being able to model check hardware designs grows proportionally with the complexity of the system models.

Chapter 5

Related works

Faults in hardware design can have serious consequences. Well known examples are faults in medical devices[1] that have caused loss of human lives. Model checking and verification, where every possible state of a model is observed before converting a design into an integrated circuit is of fundamental importance in preventing such incidents. Another example is the infamous floating-point division bug in early Intel[2] processors. The bug was rarely encountered and was discovered by a mathematician doing research in number theory[3]. Since then the use of formal methods has led to uncovering of subtle and sometimes very serious bugs.

Some recent research has been directed towards building formal semantics for BSV. Kjartansson et al. [4] have provided the formal semantics for a subset of BSV using natural semantics [8] with the aim of proving various properties of the language and reasoning about behavioural equivalence of programs. This work is closely related to what we present in this thesis.

Methodology for verification of BSV models using Spin, which is a model checking tool, has been proposed in work by Singh et al. [21]. With PROMELA [15] as the undelying model of computation, it has certain similarities with the Uppaal verification engine. Both are based on interacting processes with asynchronous communication, but the focus in this work is on how different behaviors of BSV models can be efficiently verified, aiding in faster verification. A translation schema from a subset of BSV into Promela is proposed but no automated translation between the two formalisms is undertaken as is the case in our work. The use of graphical tools is not considered either.

[1] Therac-25 investigation http://www.cs.umd.edu/class/spring2003/cmsc838p/Misc/therac.pdf
[2] http://www.intel.com/
[3] http://www.cl.cam.ac.uk/ jrh13/slides/nasa-14apr10/slides.pdf

Embedding of BSV in the PVS theorem prover[4] has been carried out by Richards et al. [19]. This allows for verification of a broader class of properties than can be achieved with model checking. An embedding strategy is introduced for several advanced language features of BSV serving as a step towards building a compiler from BSV into PVS.

Similar work to ours has been carried out in connection with other languages. Herber et al. has given a translation from SystemC into Uppaal[5] to formally verify temporal properties of SystemC designs. For this purpose the scheduler was stated as a timed automaton.

A BSV parser has been proposed[6] in connection with the development of a plug-in for an integrated development environment. A different approach from what we present here is taken in that tools are used to generate the parser from an abstract syntax tree.

In recent work by Reynisson et al. [10] an automated translation is provided from Timed Rebeca[7], an actor based modeling language, to Erlang[8].

[4] http://pvs.csl.sri.com/
[5] http://dent.cecs.uci.edu/ papers/esweek08/codes/p131.pdf
[6] Zipfel et al. http://fileadmin.cs.lth.se/esd/sw/BlueSVEP/BlueSVEP.pdf
[7] http://www.rebeca-lang.org/
[8] http://www.erlang.org/

Chapter 6

Conclusions and future work

6.1 Conclusion

In this thesis we have shown that it is possible to translate from a hardware language to the modeling language of Uppaal. This allows graphical examination of system models with the benefit of early detection of unwanted behaviour, at design time rather than in the finished hardware.

We have compared the high level hardware description language BSV and the modeling language of Uppaal. We show that similarities of their basic constructs, in particular the guarded atomic action execution model, facilitate translation between the two.

Even though BSV is not well suited for formal verification, its properties give rise to the use of external verification systems for formal reasoning about behavioural aspects such as reachability, safety, liveness and deadlock properties. Some valuable checking can be performed in BSV by placing invariants at strategically chosen places in the code, but to our knowledge no tools for BSV exist that facilitate graphical verification and model checking.

We define a simplified version of BSV, suitable for translation into Uppaal, and propose a scheme for translating from BSV into Uppaal. We furthermore provide a prototype of a tool for translating from a simplified version of BSV into the Uppaal description language.

The main contribution of this thesis is to provide means to express a hardware description as a network of timed automata.

6.2 Future work

Some enhancements to the compiler will enable examination of more complex SBSV models and reducing the size of the generated Uppaal code.

It would be quite interesting to direct the translation in reverse to what is proposed in this work (Figure 1.3), in which case a hardware description would be written in the Uppaal environment and compiled into SBSV.

A plug-in might be written and added to the Uppaal toolkit. Such a plug-in will enable a user to design hardware from a set of predefined hardware constructs in the form of automata. The system could be designed in a modular fashion, with the native tools for simulating and model checking at hand. After analyzing the model it can be compiled into SBSV. Further compilation to standard BSV will then be the last stage in converting the design to finished quality hardware.

Part I

Appendix

Appendix A

Example code listings

Below is the full code listing[1] for some of the examples discussed in the thesis.

A.1 Code for the example in Section 3.5

Listing A.1: BSV version of the Jobshop

```
1
2  package JobshopTb;
3  (* synthesize *)
4  module mkJobshopTb (Empty);
5
6    Hammer_ifc hammer <- mkHammer (5);
7    Jobber_ifc jobber <- mkJobber (5);
8
9    Reg#(int) jobs <- mkReg (0); // currently not used
10
11   rule work_hard if (hammer.isFree());
12     hammer.getHammer();
13     $display(_:: work_hard_, jobber.workHard());
14   endrule
15
16   rule rest if (!hammer.isFree());
17     $display(_:: rest_);
```

[1] Note that in the program listings the quotation mark glyphs " and ′ have been replaced with __ due to a Latex problem workaround. This must be reverted if the language description is to be opened in Uppaal.

```
18      hammer.putHammer();
19    endrule
20
21    rule work_easy;
22      $display(_:: work_easy_, jobber.workEasy());
23    endrule
24
25    rule finale (jobber.enough());
26      $display (_fired finale :: Work finished_);
27      $finish (0);
28    endrule
29
30  endmodule
31
32  // interfaces
33
34  interface Jobber_ifc;
35    method ActionValue#(int) workEasy();
36    method ActionValue#(int) workHard();
37    method Bool enough();
38  endinterface
39
40  interface Hammer_ifc;
41    method Action getHammer();
42    method Action putHammer();
43    method Bool isFree();
44  endinterface
45
46  // submodules
47
48  (* synthesize *)
49  module mkJobber #(parameter int number) (Jobber_ifc);
50
51    Reg#(int) hardjobs <- mkReg (0);
52    Reg#(int) easyjobs <- mkReg (0);
53
54    method ActionValue#(int) workEasy();
55      easyjobs <= easyjobs + 1;
56      $display(_jobber %x is working easy_, number);
57      return easyjobs;
```

```
58    endmethod
59
60    method ActionValue#(int) workHard();
61      hardjobs <= hardjobs + 1;
62      $display(_jobber %x is working hard_, number);
63      return hardjobs;
64    endmethod
65
66    method Bool enough();
67       return (easyjobs + hardjobs >= 7);
68    endmethod
69
70  endmodule
71
72  (* synthesize *)
73  module mkHammer #(parameter int init_val) (Hammer_ifc);
74
75    Reg#(Bool) hammerIsTaken <- mkReg (False);
76
77    function Action takeHammer();
78      return action
79        hammerIsTaken <= True;
80        $display(_function takeHammer : hammerIsTaken = %b_,
81                    hammerIsTaken);
82      endaction;
83    endfunction
84
85    function Action placeHammer();
86      return action
87        hammerIsTaken <= False;
88        $display(_function placeHammer : hammerIsTaken = %b_,
89                    hammerIsTaken);
90      endaction;
91    endfunction
92
93    method Action getHammer() if (hammerIsTaken == False);
94      $display(_method getHammer_);
95      takeHammer();
96    endmethod
97
```

```
98    method Action putHammer() if (hammerIsTaken == True);
99       $display(_method putHammer_);
100      placeHammer();
101   endmethod
102
103   method Bool isFree();
104      return hammerIsTaken == False;
105   endmethod
106
107 endmodule
108
109 endpackage
```

A.2 Parallel execution code from Section 4.4

Listing A.2: A rigid synchronous pipeline of registers

```
1  <?xml version=_1.0_ encoding=_utf-8_?>
2  <!DOCTYPE nta PUBLIC _-//Uppaal Team//DTD Flat System 1.1//EN_
3  _http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd_>
4  <nta>
5  <declaration>
6   broadcast chan tick;
7   int i;
8   int x1 = 10;int x1Start;
9   int x2 = 20;int x2Start;
10  int x3 = 30;int x3Start;
11  int x4 = 40;int x4Start;
12  int x5 = 50;int x5Start;</declaration>
13 <template>
14       <name>Chronometer</name>
15       <declaration>
16       void statefulBackup(int v1, int v2, int v3, int v4, int v5)
17 {
18    x1Start = v1;
19    x2Start = v2;
20    x3Start = v3;
21    x4Start = v4;
```

```
22      x5Start = v5;
23      i++;
24  }
25  </declaration>
26              <location id=_id0_ x=_-300_ y=_-300_>
27              </location>
28              <init ref=_id0_/>
29              <transition>
30                    <source ref=_id0_/>
31                    <target ref=_id0_/>
32                    <label kind=_synchronisation_ x=_-310_ y=_-490_>
33                          tick!
34                    </label>
35                    <label kind=_assignment_ x=_-570_ y=_-340_>
36                          statefulBackup(x1, x2, x3, x4, x5)
37                    </label>
38                    <nail x=_-350_ y=_-350_/>
39                    <nail x=_-350_ y=_-460_/>
40                    <nail x=_-250_ y=_-460_/>
41                    <nail x=_-250_ y=_-350_/>
42              </transition>
43        </template>
44        <template>
45              <name>M1</name>
46              <location id=_id1_ x=_-440_ y=_-448_>
47              </location>
48              <location id=_id2_ x=_-408_ y=_-560_>
49                    <committed/>
50              </location>
51              <location id=_id3_ x=_-256_ y=_-592_>
52                    <committed/>
53              </location>
54              <init ref=_id1_/>
55              <transition>
56                    <source ref=_id1_/>
57                    <target ref=_id2_/>
58                    <label kind=_guard_ x=_-424_ y=_-480_>
59                          true
60                    </label>
61                    <label kind=_synchronisation_ x=_-384_ y=_-528_>
```

```
62                      tick?
63                </label>
64                <nail x=_-440_ y=_-496_/>
65                <nail x=_-408_ y=_-496_/>
66          </transition>
67          <transition>
68                <source ref=_id2_/>
69                <target ref=_id3_/>
70                <label kind=_assignment_ x=_-384_ y=_-624_>
71                      x2 = x1Start + 1
72                </label>
73                <nail x=_-408_ y=_-592_/>
74          </transition>
75          <transition>
76                <source ref=_id3_/>
77                <target ref=_id1_/>
78                <label kind=_assignment_ x=_-232_ y=_-544_>
79                      x3 = x2Start + 1
80                </label>
81                <nail x=_-256_ y=_-448_/>
82          </transition>
83    </template>
84    <template>
85          <name>M2</name>
86          <location id=_id4_ x=_-256_ y=_-464_>
87          </location>
88          <location id=_id5_ x=_-232_ y=_-368_>
89                <committed/>
90          </location>
91          <location id=_id6_ x=_-432_ y=_-512_>
92                <committed/>
93          </location>
94          <init ref=_id4_/>
95          <transition>
96                <source ref=_id4_/>
97                <target ref=_id5_/>
98                <label kind=_guard_ x=_-248_ y=_-448_>
99                      true
100               </label>
101               <label kind=_synchronisation_ x=_-210_ y=_-450_>
```

```
102                         tick?
103                   </label>
104                   <nail x=_-256_ y=_-408_/>
105                   <nail x=_-232_ y=_-408_/>
106             </transition>
107             <transition>
108                   <source ref=_id5_/>
109                   <target ref=_id6_/>
110                   <label kind=_assignment_ x=_-368_ y=_-296_>
111                         x4 = x3Start + 1
112                   </label>
113                   <nail x=_-232_ y=_-304_/>
114                   <nail x=_-432_ y=_-304_/>
115             </transition>
116             <transition>
117                   <source ref=_id6_/>
118                   <target ref=_id4_/>
119                   <label kind=_assignment_ x=_-384_ y=_-544_>
120                         x5 = x4Start + 1
121                   </label>
122                   <nail x=_-256_ y=_-512_/>
123             </transition>
124       </template>
125       <system>
126             m2 = M2();
127             m1 = M1();
128             t = Chronometer();
129             system t, m1, m2;
130       </system>
131 </nta>
```

Appendix B

Miscellaneous

B.1 Code for the example in Section 2.2.1

Listing B.1: Coffee machine

```
<?xml version=_1.0_ encoding=_utf-8_?>
<!DOCTYPE nta PUBLIC _-//Uppaal Team//DTD Flat System 1.1//EN_
_http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd_>
<nta>
<declaration>// Place global declarations here. //
      chan coin, coffee, pub;
      int cups = 0;
      int pubs = 0;
      int coins = 0;
</declaration>
<template>
            <name x=_5_ y=_5_>Student</name>
            <declaration>// Place local declarations here.
            </declaration>
            <location id=_id0_ x=_-56_ y=_-32_>
            </location>
            <location id=_id1_ x=_-224_ y=_168_>
            </location>
            <location id=_id2_ x=_-224_ y=_40_>
            </location>
            <location id=_id3_ x=_-224_ y=_-80_>
            </location>
```

```
23              <location id=_id4_ x=_-224_ y=_-192_>
24              </location>
25              <init ref=_id4_/>
26              <transition>
27                    <source ref=_id0_/>
28                    <target ref=_id4_/>
29              </transition>
30              <transition>
31                    <source ref=_id0_/>
32                    <target ref=_id4_/>
33                    <label kind=_synchronisation_ x=_-40_ y=_-136_>
34                          pub!
35                    </label>
36                    <nail x=_-56_ y=_-192_/>
37              </transition>
38              <transition>
39                    <source ref=_id1_/>
40                    <target ref=_id0_/>
41                    <nail x=_-56_ y=_168_/>
42              </transition>
43              <transition>
44                    <source ref=_id2_/>
45                    <target ref=_id1_/>
46                    <label kind=_synchronisation_ x=_-284_ y=_89_>
47                          coffee?
48                    </label>
49              </transition>
50              <transition>
51                    <source ref=_id3_/>
52                    <target ref=_id2_/>
53              </transition>
54              <transition>
55                    <source ref=_id4_/>
56                    <target ref=_id3_/>
57                    <label kind=_synchronisation_ x=_-284_ y=_-151_>
58                          coin!
59                    </label>
60              </transition>
61        </template>
62        <template>
```

```
63              <name>Observer</name>
64              <location id=_id5_ x=_-168_ y=_200_>
65                    <name x=_-192_ y=_224_>complain</name>
66              </location>
67              <location id=_id6_ x=_-168_ y=_32_>
68              </location>
69              <location id=_id7_ x=_-168_ y=_-144_>
70              </location>
71              <init ref=_id7_/>
72              <transition>
73                    <source ref=_id6_/>
74                    <target ref=_id5_/>
75              </transition>
76              <transition>
77                    <source ref=_id6_/>
78                    <target ref=_id7_/>
79                    <label kind=_guard_ x=_-128_ y=_2_>
80                          cups &lt; pubs - 2
81                    </label>
82                    <nail x=_32_ y=_32_/>
83                    <nail x=_32_ y=_-144_/>
84              </transition>
85              <transition>
86                    <source ref=_id7_/>
87                    <target ref=_id6_/>
88                    <label kind=_synchronisation_ x=_-228_ y=_-71_>
89                          pub?
90                    </label>
91              </transition>
92        </template>
93        <template>
94              <name>Machine</name>
95              <location id=_id8_ x=_-368_ y=_24_>
96              </location>
97              <location id=_id9_ x=_-176_ y=_144_>
98              </location>
99              <location id=_id10_ x=_-176_ y=_-40_>
100             </location>
101             <location id=_id11_ x=_-176_ y=_-200_>
102             </location>
```

```
103             <init ref=_id11_/>
104             <transition>
105                   <source ref=_id8_/>
106                   <target ref=_id11_/>
107                   <label kind=_synchronisation_ x=_-424_ y=_-120_>
108                         coffee!
109                   </label>
110                   <label kind=_assignment_ x=_-424_ y=_-96_>
111                         cups++
112                   </label>
113                   <nail x=_-368_ y=_-200_/>
114             </transition>
115             <transition>
116                   <source ref=_id9_/>
117                   <target ref=_id8_/>
118                   <nail x=_-368_ y=_144_/>
119                   <nail x=_-368_ y=_112_/>
120             </transition>
121             <transition>
122                   <source ref=_id9_/>
123                   <target ref=_id11_/>
124                   <label kind=_synchronisation_ x=_16_ y=_-64_>
125                         coffee!
126                   </label>
127                   <label kind=_assignment_ x=_16_ y=_-40_>
128                         cups++
129                   </label>
130                   <nail x=_-8_ y=_144_/>
131                   <nail x=_-8_ y=_-200_/>
132             </transition>
133             <transition>
134                   <source ref=_id10_/>
135                   <target ref=_id9_/>
136             </transition>
137             <transition>
138                   <source ref=_id11_/>
139                   <target ref=_id10_/>
140                   <label kind=_synchronisation_ x=_-236_ y=_-135_>
141                         coin?
142                   </label>
```

```
143                     <label kind=_assignment_ x=_-236_ y=_-120_>
144                           coins++
145                     </label>
146               </transition>
147         </template>
148         <system>
149               machine = Machine();
150               student = Student();
151               observer = Observer();
152               system student, machine, observer;
153         </system>
154   </nta>
```

B.2 Scheduling rules within a module

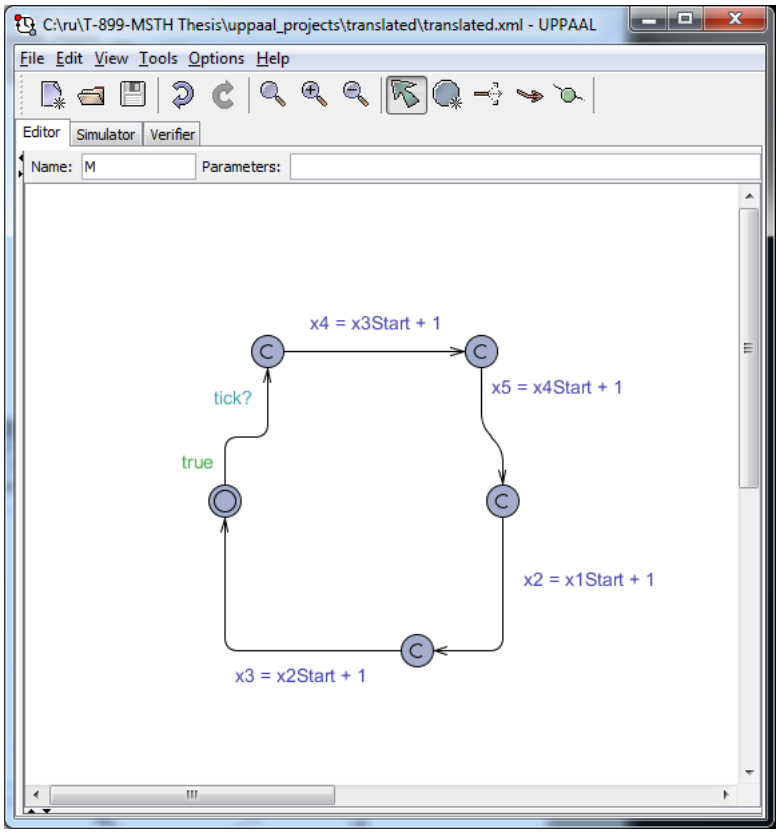


Figure B.1: Rescheduled rules running in parallel

In a given state, a module chooses one rule for which the guard evaluates to true and applies the associated action. If more than one guard is true, a non-deterministic choice is made. The translation (Section 3.4) carries this non-deterministic transition relation for module instances [19] over to Uppaal and the choice of transition to be performed is left to a scheduler, as is the case in BSV where the compiler decides the order of rules within a clock cycle [16, ch. 6.2.3] [19, section. 2.1]. We present here a proposal for how a scheduler might work in principle in Uppaal.

In Section 4.4 a rigid synchronous pipeline of registers written in SBSV was translated into Uppaal description language. Because the modules in the example are two, the translation results in two automata that synchronize with a clock.

If a module were to contain two or more rules it would translate to one automaton with as many transitions as there are rules. For those transitions to be performed in parallel, the atomaton has to be modified slightly by a scheduler. This is analogus to the situation in BSV when several rules inside a module are to be fired in parallel as discussed in Section 3.4.

A hypothetical scheduler that modifies the example in Section 4.4 simply combines the rules into one aggregated transition (Figure B.1).

Because of the implementation introduced in Section 3.4, where we copy the values of the registers at the start of a clock cycle, and allow a read from the registers at all times, all the registers are updated in one clock cycle.

Bibliography

[1] Rishiyur S. Nikhil and Kathy R. Czeck *BSV by Example*

[2] Uppaal *uppaal.org*

[3] Luca Aceto, Anna Ingólfsdóttir, Kim G. Larsen and Jiři Srba *Reactive Systems Modelling, Specification and Verification*

[4] Oddur Óskar Kjartansson *On the Formal Semantics of Bluespec System Verilog*

[5] James C. Hoe and Arvind *Hardware Synthesis from Term Rewriting Systems*

[6] Alur R. and Dill D. L. *A theory of timed automata*

[7] Edsger W. Dijkstra *Guarded Commands, nondeterminacy and Formal Derivation of Programs*

[8] Hanne Riis Nielson and Flemming Nielson *Semantics with Applications: A formal Introducion*

[9] Robin Milner *Communication and Concurrency*

[10] Árni Hermann Reynisson *Timed Rebeca: Refinement and Simulation*

[11] Michael Huth and Mark Ryan *Logic in computer science*

[12] Adele Goldberg and David Robson *Smalltalk-80: The Language*

[13] Ken Arnold, James Gosling and David Holmes *The Java Programming Language*

[14] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman *Compilers: Principles, Techniques, and Tools*

[15] Gerard J. Holzmann *Spin Model Checker: Primer and Reference Manual*

[16] *Bluespec SystemVerilog Reference Guide Revision: 17 January 2012*

[17] *Bluespec SystemVerilog User Guide Revision: 17 January 2012*

[18] JDOM *jdom.org*

[19] Dominic Richards and David Lester *A monadic approach to automated reasoning for Bluespec SystemVerilog*

[20] Gérard Berry *A hardware implementation of pure Esterel*

[21] Gaurav Singh and Sandeep K. Shukla *Verifying Compiler Based Refinement of Bluespec*

[22] Frits Vaandrager *A First Introduction to Uppaal*

[23] Shallow SBSV translator *xp-dev.com/svn/shallow*

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539