AUTOMATIC MAPPING OF STATECHART INTO VERILOG

TRAN VU VIET ANH

(B.Sc, Hanoi University of Science, Vietnam)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

I express my deep gratitude to my parents.

I would like to give sincere thanks to my advisor A/P Chin Wei Ngan for his invaluable guidance, crucial suggestions, and support to my study and research. With his understanding and his willing to help, I have overcome many difficulties in last two years.

I thank also Dr. Qin Shengchao for his valuable and critical comments that helped to improve the preciseness of the system and the quality of the report. Through the discussions, he helps me to figure out many good points which I misunderstand. I greatly appreciate the kindness of A/P Khoo Siau Cheng. Thanks the members of Programming Language System Lab and my friends.

Last but not least, special thanks to my girlfriend, Thanh Van, for her understanding, support, and boosting my morale when I'm tired or bored.

Contents

Summary

This thesis establishes a connection between statechart and Verilog programming language. Statechart (invented by David Harel) is a powerful visual formalism for specifying discrete event systems and is a significant specification language. Statechart diagrams capture the behavior of entities capable of dynamic behavior by specifying the responses to possible event instances. Verilog is widely used for hardware description in industry. There are number of works on both statechart and Verilog. However, the translation from statechart to Verilog is still unexplored. In this work, we shall propose a system to map each source statechart specification into corresponding target Verilog code. We demonstrate our work through an implemented system and some examples.

Our implementation is divided into two parts: a statechart editor and a mapping program. The editor, called Statechart_E, is exactly a stencil that has been built as an add-on to Microsoft Visio. The mapping program, called

AMSV (Automatic Mapping of Statechart into Verilog), is written in Java. Details of the algorithms and implementation are discussed in chapters 3 and 4 of this thesis.

Background of statechart and its specification are presented in chapter 2. In this chapter, we shall also discuss the syntax and algebraic laws of Verilog. This chapter sets the scene for the rest of the work by introducing the basic knowledge and key notations need. Chapter 5 presents two case studies to illustrate our work. In chapter 6, we introduce the concrete Verilog program and discuss a possible solution to transform abstract Verilog to concrete Verilog.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Statechart

In this thesis, we have used *Statechart* [13–16] (detail in section 2.1) for systems specification. Statechart is a powerful visual formalism for specifying discrete event systems. It retains the visual and intuitive appeal inherent for state transition systems and extends these systems in the following ways:

- Hierarchy
- Orthogonal (concurrency)
- Broadcast communication

Fig. 1.1 shows an example of a statechart with eight states. Here, P0 is an `And`-state, with two inner `Or`-states; P1 and P2. P3 is the default sub–state of P1, P5 is the default sub–state of P2. Altogether, there are six transitions in the statechart.

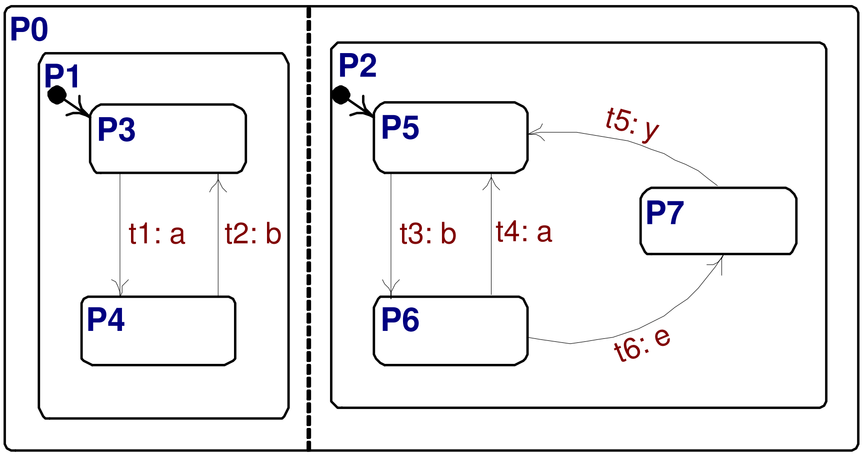


Figure 1.1: An example of a statechart.

Statechart descriptions can be readily simulated and translated to hardware description languages such as Verilog. The relevance of statecharts as inputs to our developed tools will become clear later.

1.2 Verilog

Verilog [21, 32] is a widely used language for hardware description in industry [6, 12, 25, 26] and also in research. Verilog is used to model the structure and behaviour of digital systems ranging from simple hardware building block to complete systems. Verilog semantics is based on the scheduling of events and the propagation of changes. One early attempt to investigate the semantics of Verilog is the work of Gordon [12] which explains how top-level modules can be simulated.

A Verilog program (or specification, as it is more frequently referred to) is a description of a device or process rather similar to a computer program written in C or Pascal. However, Verilog also includes constructs specifically chosen to describe hardware. One major difference from a language like C is that Verilog allows processes to run in parallel. This is obviously very desirable if one is to exploit the inherently parallel behaviour of hardware. In this project and thesis, we will make use of abstract Verilog [26, 37], that is described in the next chapter.

Quoting from Gordon [12]: "Verilog is a relatively simple real–world language in need of theoretical support. It poses a variety of interesting semantics and logical challenges ranging from routine applications of standard techniques (e.g. formalizing the simulation cycle) to hard theoretical problems (e.g. developing a theory of behavioural congruence)"

Moreover, what started initially as a proprietary hardware modelling language by Gateway Design Automation Inc. around 1984 and first used in 1985 and was extended substantially through 1987. Nowadays, there are a large numbers of Verilog users and designers. Listed by Google search engine on 20th November, 2003, there are more than 300,000 links related to Verilog. This indicates a huge number of works related to Verilog around the world.

1.3 Motivation: Bridge the gap between Statecharts and Verilog

As introduced in last two subsections, Statecharts is a visual formalism catering for high-level behaviourial specification of embedded systems. Its hierarchical structure and orthogonal features make the system specification compact and intuitive to understand. It is a very good candidate for executable specification in system design. Moreover, the formal semantics of Statecharts has been extensively investigated [13–16, 17, 44] in recent years. Some works also attempt to provide tools for formal verification of Statecharts specifications [9, 28, 42].

On the other hand, Verilog is a hardware description language that has been widely used by hardware designers. Its rich features make it a good candidate for low–level system specifications. The formal semantics of Verilog was first given by Gordon [12] in terms of simulation cycles. It has been thoroughly investigated afterwards [18, 19].

As the advantages of Statecharts and Verilog in embedded system design process are complementary to each other, a natural question that can be raised is, can we make use of both of them in system design? That is, can we use Statecharts as the high level specification, while use Verilog as the low level description? This question has motivated our work and this thesis shall provide a positive answer by bridging the gap between Statecharts and Verilog. The compilation from Statecharts to Verilog can be embedded into the hardware/software co-specification process. A mapping algorithm will be given in the following chapters, where the soundness has been given in Qin and Chin [37].

1.4 Layout of the thesis

The thesis is organized as follows:

- **Chapter 2** In this chapter, we introduce some details about statecharts, Verilog language and key notations used.
- **Chapter 3** This chapter explores the mapping algorithm that is used to translate statecharts to abstract Verilog.
- **Chapter 4** This chapter presents the implementation of our system. We discuss how we use the mapping algorithm in our system.
- **Chapter 5** This chapter discusses some case studies that are used to illustrate the algorithm and our results.
- **Chapter 6** we discuss the transformation from Verilog to concrete Verilog in this chapter.
- **Chapter 7** The last chapter concludes with a discussion on future work.

Chapter 2

Preliminaries

This chapter sets the scene for the rest of the work by introducing basic knowledge and notation used. Section 2.1 presents more detail of Statechart and its operational semantics. Both states and transitions are formally defined in this section. Section 2.2 introduces valid laws of Verilog language, which we use in our work. The last section summarizes key notations used in this thesis.

2.1 Statechart

2.1.1 A formal syntax of statechart

Statecharts is specification language derived from finite-state machines. The language is rather rich in features including state hierarchy and concurrency. Transitions can perform nontrivial computations unlike finite-state machines where they contain at most input/output pairs. In this section we will describe Statecharts presented by David Harel [13–15].

Statechart diagrams capture the behaviour of entities capable of dynamic behaviour by specifying their responses to the event occurrences. Typically, it is used for describing the behaviour of classes, but statecharts may also describe the behaviour of other model entities such as use cases, actors, subsystems, operations, or methods.

We use a simple textual representation of Statecharts, while our system can automatically translate a graphical representation to the textual representation. The statecharts language we adopt has some features that are not present in UML statecharts. For example, broadcast communication is supported in our language but not in UML statecharts. It is also possible to adopt XMI as a representation language for our Statecharts language. This is left for future consideration."

As already mentioned in previous chapter, Statecharts is extensible by hierarchy, orthogonality or broadcast communication. In this thesis, we use the formal syntax of statechart from [14] and [37]. The syntax of Statecharts formula is defined as follows (quoting from [37]):

$\mathcal{S}$: a set of names used to denote Statecharts. This is expected to be large enough to prevent name conflicts.

Π_e : a set of all abstract events (signals). We also introduce another set $\overline{\Pi}_e$ to denote the set of negated counterparts of events in Π_e , i.e. $\overline{\Pi}_e =_{df} \{\overline{e} \mid e \in \Pi_e\}$, where $\overline{e}$ denotes the negated counterpart of event e, and we assume $\overline{\overline{e}} = e$.

Π_a : a set of all assignment actions of the form $v = exp$.

$\sigma : Var \rightarrow Val$ is the valuation function for variables, where Var is the set of all variables, Val is the set of all possible values for variables. A snapshot for variables $\overline{v}$ is $\sigma(\overline{v})$.

$\mathcal{T}$: a set of transitions, which is a subset of $\mathcal{S} \times 2^{\Pi_e \cup \overline{\Pi}_e} \times 2^{\Pi_e \cup \Pi_a} \times \mathcal{B}_e \times \mathcal{S}$, where $\mathcal{B}_e$ is the set of boolean expressions.

A term-based syntax of statecharts was introduced in [37] and [28–31]. We re-introduce it here for the benefit of the reader. The set $\mathtt{SC}$ is a set of State-charts terms that is constructed by the following inductively defined functions.

$\mathtt{Basic} : \mathcal{S} \rightarrow \mathtt{SC}$

$\mathtt{Basic}(s) =_{df} |[s]|$

$\mathtt{Or} : \mathcal{S} \times [\mathtt{SC}] \times \mathcal{T} \rightarrow \mathtt{SC}$

$\mathtt{Or}(s, [p_1, ..., p_l, ..., p_n], p_l, T) =_{df} |[s : [p_1, ..., p_l, ..., p_n], p_l, T]|$

$\mathtt{And} : \mathcal{S} \times 2^{\mathtt{SC}} \rightarrow \mathtt{SC}$

$\mathtt{And}(s, \{p_1, ..., p_n\}) =_{df} |[s : \{p_1, ..., p_n\}]|$

Note that:

– $\mathtt{Basic}(s)$: denotes a basic statechart named s.

– $\mathtt{Or}(s, [p_1, ..., p_l, ..., p_n], p_l, \mathcal{T})$: represents an $\mathtt{Or}$-statechart with a set of sub-states $\{p_1, ..., p_n\}$, where p_1 is the default sub-state, p_l is the current active sub–state, T is composed of all possible transitions among immediate sub-states of s.

– $\mathtt{And}(s, \{p_1, ..., p_n\})$ is an $\mathtt{And}$-statechart named s, which contains a set of orthogonal (concurrent) sub-states $\{p_1, ..., p_n\}$.

2.1.2 State

A state is a condition during the life of an object or an interaction during which it satisfies some conditions, performs actions, or waits for some events. A composite statechart is a state that, in contrast to a simple state, can be decompounded into smaller Statecharts (composite states and their notation are described in more detail latter.) Conceptually, an object remains in a state for an interval of time. However, the semantics allows for modelling to "flow-through" states in an instantaneous manner, as well as transitions that are not instantaneous.

In the diagram, a state is shown as a rectangle with rounded corners. Each state must have at least a name, and it may contain other information like: entry/exit, activities, internal transitions, sub-states, deferred events, etc. Fig. 2.1 shows an example of a basic-state with the name `State`.

Figure 2.1: Example of a state.

2.1.3 Event

An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition. Events may be of several kinds (not necessarily exclusive).

- A designated condition becoming true (described by a Boolean expression) results in a change event instance. The event occurs whenever the value of the expression changes from false to true. Note that this is different from a guard condition. A guard condition is evaluated once whenever its event fires. If it is false, then the transition does not occur and the event is lost.

- The receipt of an explicit signal from one object to another results in a signal event instance. It is denoted by the signature of the event as a trigger on the transition.

- The receipt of a call for an operation implemented as a transition by an object is called a call event instance.

- The passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date/time is called a TimeEvent.

The event declaration has scope within the package it appears in and may be used in state diagrams for classes that have visibility inside the package. Take note that events are not local to class.

2.1.4 Transition

A simple transition is a relationship between two states indicating that an object in the first state (source state) will enter the second state (target state). Furthermore, it will perform specific actions when the event occurs provided that certain specified conditions are satisfied. During such a change of state, the transition is said to “fire”. The trigger for a transition is the occurrence of the event labelling for the transition. The event may have parameters, which are accessible by the actions specified on the transition as well as in the corresponding exit and entry actions associated with the source and target states respectively. Events are processed one at a time. If an event does not trigger any transition, it is discarded. If it can trigger more than one transition within the same sequential region (i.e., not in different concurrent regions), only one will fire. If these conflicting transitions are of the same priority, an arbitrary one is selected and triggered.

A transition is shown as a solid line originating from the source state and terminated by an arrow on the target state. It may be labeled by a transition string that has the following general format:

Name: event signature / action–expression [condition]

where, *Name* is the transition name. The *event signature* (may contain several events, separated by a comma) describe events with its arguments and a transition can be taken only if its event occurs. The *action-expression* is executed if and when the transition fires. The *condition* is a Boolean expression written in terms of the parameter of the triggering event and attributes.

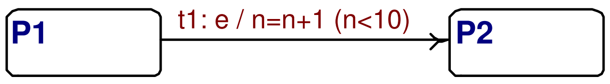


Figure 2.2: Example of a transition between two states P1 and P2.

Fig. 2.2 is an simple example of a transition from source state P1 to target state P2, where $t1$ is transition name. Transition $t1$ will be fired if event e occurs and action $n = n + 1$ will be executed. $n < 10$ is condition for the transition.

The syntax of a transition with its arguments (source and target states, *Name*, *event signature*, *action-expression*, and *condition*) is:

$$\begin{aligned} Name = \ & \langle source\ state, event\ signature, action - expression, \\ & condition, target\ state \rangle \end{aligned}$$

For example, $t1$ in Fig. 2.2 will be written as:

$$t1 = \ \langle \mathrm{P1}, e, n = n + 1, n < 10, \mathrm{P2} \rangle$$

More complex transitions are transitions which normally do not connect between two states of the same parent, for example transitions from/to concurrent states or composite states. Transitions of concurrent states may have multiple source states and target states. It represents synchronization and/or a splitting of control into concurrent threads without concurrent sub-states. Those transitions are enabled when all of the source states are occupied. After a transition fires, all of its destination states are occupied. Transitions in composite states are drawn to the boundary of composite states. The entry action is always performed when a state is entered from outside.

Fig. 2.3 shows an example of concurrent transitions and composite transitions. Here, ($t1$, $t3$), ($t2$, $t4$) are concurrent transitions, and $t7$, $t8$ are composite transitions.

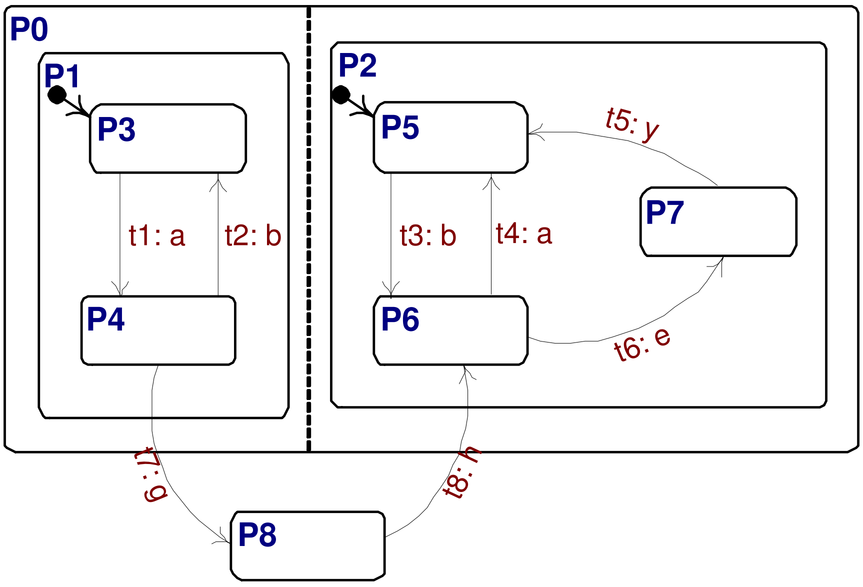


Figure 2.3: Example of complex transitions.

2.1.5 State hierarchy

A statechart contains some sub-states, and these sub-states may be other statecharts (contain states inside). In this case we have a hierarchic statechart. Fig. 2.4 shows an example of a hierarchic statechart. It describes a tape-recorder (named `Tape_recorder`) with three states; namely `Stop`, `Record`, and `Control`. The `Tape_recorder` is an `Or`-state with the following syntax description:

Tape_recorder = |[s, [Stop, Record, Control], Stop, T]|

where, $T = \{t1, t2, r3, t4\}$ is a set of transitions within `Tape_recorder` and `Stop` is the default sub-state. The `Control` state is a sub-state of `Tape_recorder`. And it is an `Or`-state with `Play` and `FF` states inside.

In fact, it is easy to read the statechart in Fig. 2.4 as a hierarchical version of the statechart in Fig. 2.5. In the second version, we removed the `Control`

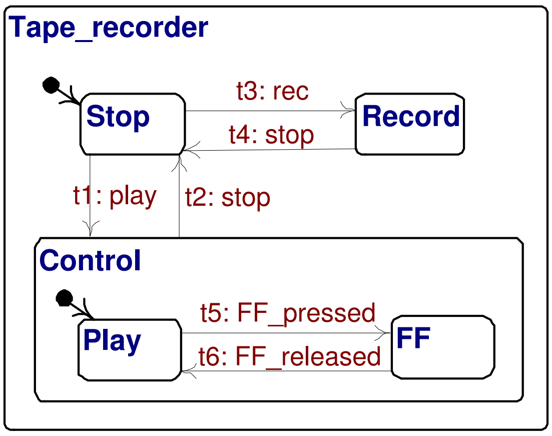


Figure 2.4: Statechart of a tape-recorder with hierarchy.

state and rewire it with transitions, such as $t1, t2$ and add a new transition $t2\,a$. The net effect of two statecharts is no difference. However, according to the semantics of statecharts in [15], when taking a transition, we must allow `Basic`-states to be entered. For example, in Fig. 2.5, all states have transitions to `Stop` with event *stop*. We called this a compound transition.

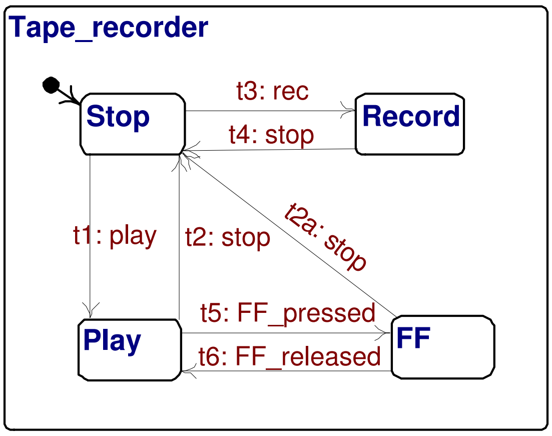


Figure 2.5: Statechart of a tape-recorder, flattened version.

Hierarchical statecharts are very common in real systems. It can handle more complex system and also contain concurrent states (presented in next subsection). With a more sophisticated statechart, the source and target states of transitions need not be at the same level (same parent state). To deal with this problem, we can calculate the *depth* of a state, called *or-depth*, with a function that calculates the depth of the path along its transitions and active states. The *or-depth* of a `Basic`-state is 0, because `Basic`-state contains no sub-state. An `Or`-state of formula $|[s : [p_1, ..., p_n], p_l, T]|$, is *or-depth*$(p_l)$+1.

2.1.6 Concurrency

Statecharts have constructs to express concurrency. A composite state (or called an `And`-state) is decomposed into two or more orthogonal sub-states. And each orthogonal sub-state may have an initial and a final state. A transition to this `And`-state represents a transition into all initial states. For example, consider an extension of the tape recorder which provides a search facility. A user can get the tape to advance forward or backward even while the tape recorder is playing or recording. The statechart is depicted in Fig. 2.6. States with concurrently executed components are called `And`-states. In this statechart we can have more than one compound transitions executing concurrently, provided they reside in concurrent states; such as stop.

Statechart `Tape_recorder` is an `And`-state with two orthogonal sub-states, `Control` and `Search`. The syntax of `Tape_recorder` is:

Tape_recorder = |[s: {Control, Search}]|

where, `Stop` and `Idle` are initial states of `Control` and `Search` regions. If

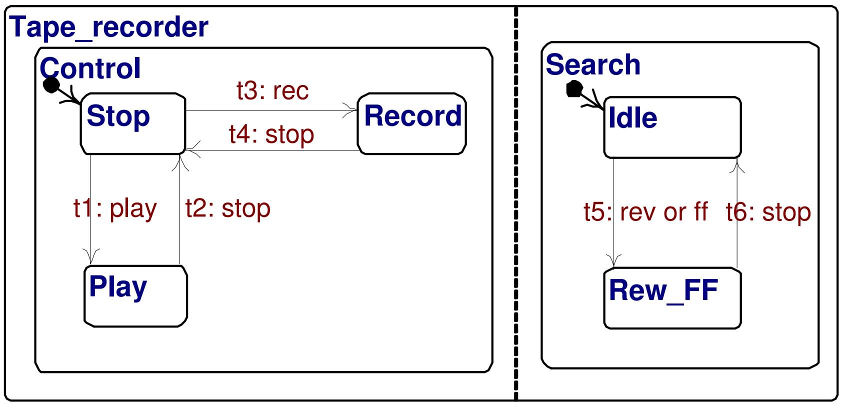


Figure 2.6: Statechart of a tape-recorder with the search function.

there exists any transition to `Tape_recorder`, it means that this transition will go to `Stop` and `Idle` simultaneously.

The *or-depth* function is also defined for the `And`-states. However, *or-depth* value of `And`-states with formula $|[s : \{p_1, ..., p_n\}]|$ is always 1.

In the next section, we discussed about Verilog programming language, a target language for our statecharts. Verilog can support parallel processes, but only at the top level. Hence, if a given statechart has `And`-states inside, we have to deal with it via expansion rules. These rules are discussed in section 2.2.3.

Another example of `And`-state was shown earlier in Fig. 1.1, where P0 is also a concurrent statechart. If concurrent states were not available, we would have to represent the statechart of Fig. 1.1 with a more complex finite state machine (FSM) in Fig. 2.7.

In this thesis we use sub-state interchangeables as children term of `Or`-state. Correspondingly, we use children and region of `And`-state interchangeably. For statecharts used this thesis, we shall assume that each `And`-state will have at

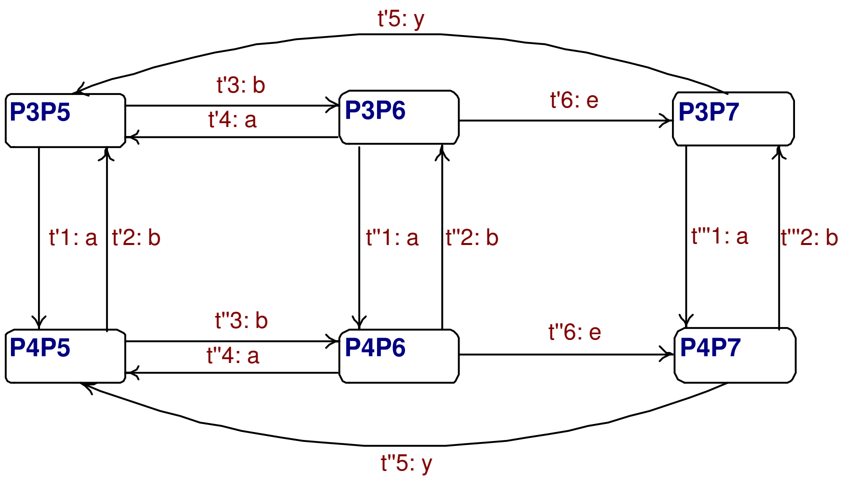


Figure 2.7: The flat representation of Fig. 1.1.

least two regions. Furthermore, each region shall be an `Or`-state.

2.1.7 Textual representation

We shall take the textual representation of statecharts as input data for our mapping program. The format of a textual representation of a statechart follow the syntax of states and transitions presented in previous sub-sections. For example, the textual representation of statechart in Fig. 1.1 is:

```
P0 = |[ S1: { P1, P2 } ]|
P1 = |[ S2: [ P3, P4 ], P3, { t1, t2 } ]|
P3 = |[ S3 ]|
P4 = |[ S4 ]|
P2 = |[ S5: [ P5, P6, P7 ], P5, { t3, t4, t5, t6 } ]|
P5 = |[ S6 ]|
P6 = |[ S7 ]|
P7 = |[ S8 ]|

t1 = < P3, { a }, {  }, true, P4 >
t3 = < P5, { b }, {  }, true, P6 >
t2 = < P4, { b }, {  }, true, P3 >
```

```
t4 = < P6, { a }, {  }, true, P5 >
t5 = < P7, { y }, {  }, true, P5 >
t6 = < P6, { e }, {  }, true, P7 >
```

In chapter 5, we shall show more complex examples.

2.2 Verilog

This section gives a brief overview of Verilog. Verilog was introduced around 1984 by Gateway Design Automation Inc, and the first used in 1985. However, the formal semantics of Verilog has not well-studied until recent years. All essential topics will be treated in some depth, and this obviously includes the full language that will be treated formally later. The semantics of Verilog is usually given in terms of how a simulator should behave and there are many previous efforts which use this approach. Until recently, the semantics of Verilog is formally introduced in the works of Gordon [12] and He [25, 26]. In our project we shall use the operational semantic of Verilog, which was introduced in [26] and [37]. The next sub-section gives an algebraic presentation of Verilog. This is followed by a description on parallel expansion laws for parallel composition.

2.2.1 Abstract Verilog language

In this project, we shall use a simple version of Verilog presented in [26, 37]. This is based on an algebraic model of Verilog. This more abstract version of Verilog can be used to express designs at various levels of hardware behaviour. Such an abstract design can be gradually refined into an equivalent counterpart in the

Verilog HDL which can provide a closer match to the underlying architecture of the hardware. This process may be repeated until the design is at a sufficiently lower level such that the hardware device can be synthesised from it. There are two main features in abstract Verilog that are not present in Verilog HDL, namely guarded choice extension and recursion. The translation from general guarded choices to parallel composition in normal Verilog is achievable, although nontrivial. The conversion of recursion to iteration is harder but there exists standard conversion techniques to realise some subsets of them. Furthermore, for bounded recursion, it is possible to inline the abstract Verilog code so as to remove recursion.

A Verilog program can be a parallel or a sequential process, but only parallel process may contain sequence processes, not vice-versa. Here are some categories of syntactic elements:

1. Parallel process

   we have:

   $P \ ::= \ S \ \mid \ P \parallel \ P$

   where, S is a sequential process.

2. Sequential process can be formally described as following

$$\begin{aligned} S \ ::= \ & PC \text{ (primitive command) } \mid S; S \text{ (sequence composition)} \\ & \mid \ s \ \lhd \ b \ \rhd \ S \text{ (condition) } \mid \ b \ * \ S \text{ (iteration)} \\ & \mid \ (b\&g \ S) \ [] \ ... \ [] \ (b\&g \ S) \text{ (guarded choice) } \mid \ fix \ X \bullet S \text{ (recursion)} \end{aligned}$$

where, b is boolean condition, and

$$
\begin{array}{lll}
PC & ::= & \mathit{skip} \mid \mathit{sink} \mid \bot \mid\ \rightarrow\ \eta \text{ (output event)} \mid v\ =\ ex \text{ (assignment)} \\
g & ::= & \rightarrow\ \eta \mid @(x = v) \text{ (assignment guard))} \\
 & & \mid \#1 \text{ (time delay)} \mid eg \text{ (event control)} \\
eg & ::= & \eta \mid eg\ \&\ eg \mid eg\ \&\ \neg eg \\
\eta & ::= & \uparrow v \text{ (value rising)} \mid\ \downarrow v \text{ (value falling)} \mid \underline{e} \text{ (a set of abstract events)}
\end{array}
$$

Recall that a Verilog program can only be a parallel processes at the top level, a sequential process cannot contain a parallel processes. However, most real systems contain many parallel processes possibly organised hierarchically. To solve this restriction, we shall use an expansion rule to change parallel code into guarded choice in subsection 2.2.3.

Here are some examples of abstract Verilog:

– $(e\ \&\ (\rightarrow f)\ \mathit{sink})\ []\ (g\ \&\ (\rightarrow h)\ \mathit{sink})$

– $\mu X \bullet (e\ (f\ X)\ )$

– $(a\ \&\ (\rightarrow e)\ \mathit{sink})\ \|\ (b\ \&\ (\rightarrow f)\ \mathit{sink})$

2.2.2 Guarded choice

To facilitate equational reasoning, we shall add a guarded choice construct to the language as we showed in the syntax. This takes as arguments a number of guarded processes. A guarded process is a guard and a process $(g\ P)$.

$$[]\{\ g_1\ P_1, ..., g_n\ P_n\}$$

Guards may be simple tests:

$$@(e),\ \#(1),\ e,\ \tau$$

or *composite* tests:

$$@(\eta_1 \; or \; ... \; or \; \eta_n), \;\; b\&e, \;\; b\&\tau$$

where b denotes a Boolean expression.

2.2.3 Parallel expansion

Parallel processes in Verilog can interact with each other via shared variables. Furthermore, the parallel operator is associative and symmetric, and has skip as its unit and $\perp$ as its zero. This properties are captured by the following rules:

1. $P \parallel Q \; = \; Q \parallel P$

2. $(P \parallel Q) \parallel R = \; P \parallel (Q \parallel R)$

3. $P \parallel \texttt{skip} \; = \; P$

4. $P \parallel \perp \; = \; \perp$

   The following expansion rules permit us to convert a parallel construct into sequential one in term of guarded choice. In program of algebra of Verilog, guarded choice plays a role of head normal form [26]. It can be regarded as a textual representation of the corresponding labeled transition system.

5. Let $P \; = \; P_1 \parallel P_2$

   where $P_1 \; = \; g_1 \, P_1'$ and $P_2 \; = \; g_2 \, P_2'$. We then have:

$$P = \begin{pmatrix} g_1 \ \& \ g_2 \ (P_1' \parallel P_2') \\ [] \ g_1 \ \& \ \neg g_2 \ (P_1' \parallel g_2 \ P_2') \\ [] \ g_2 \ \& \ \neg g_1 \ (P_2' \parallel g_1 \ P_1') \end{pmatrix}$$

The more general expansion laws are:

6. Let $P = P_1 \parallel P_2$

   where, $P_1 = []_{i=1}^{n} \ g_{1_i} \ P_{1_i}$ and $P_2 = []_{j=1}^{m} \ g_{2_j} \ P_{2_j}$ and if let $g_1 = g_{1_1}$ &...& g_{1_n}, $g_2 = g_{2_1}$ &...& g_{2_m}, we then have:

$$P = \begin{pmatrix} []_{i=1,j=1}^{n,m} \ g_{1_i} \ \& \ g_{2_j} \ (P_{1_i} \parallel P_{2_j}) \\ [] \ []_{i=1}^{n} \ g_{1_i} \ \& \ \neg g_2 \ (P_{1_i} \parallel []_{j=1}^{m} \ g_{2_j} \ P_{2_j}) \\ [] \ []_{j=1}^{m} \ g_{2_j} \ \& \ \neg g_1 \ (P_{2_j} \parallel []_{i=1}^{n} \ g_{1_i} \ P_{1_i}) \end{pmatrix}$$

7. Let $P = P_1 \parallel P_2 \parallel P_3$, where, $P_1 = g_1 \ P_1'$, $P_2 = g_2 \ P_2'$, and $P_3 = g_3 \ P_3'$ and if expand $P' = P_2 \parallel P_3$ first, then $P = P_1 \parallel P'$, we then have:

$$P = \begin{pmatrix} g_1 \ \& \ g_2 \ \& \ g_3 \ (P_1' \parallel P_2' \parallel P_3') \\ [] \ g_1 \ \& \ g_2 \ \& \ \neg g_3 \ (P_1' \parallel P_2' \parallel g_3 \ P_3') \\ [] \ g_1 \ \& \ \neg g_2 \ \& \ g_3 \ (P_1' \parallel g_2 \ P_2' \parallel P_3') \\ [] \ g_1 \ \& \ \neg g_2 \ \& \ \neg g_3 \ (P_1' \parallel g_2 \ P_2' \parallel g_3 \ P_3') \\ [] \ \neg g_1 \ \& \ g_2 \ \& \ g_3 \ (g_1 \ P_1' \parallel P_2' \parallel P_3') \\ [] \ \neg g_1 \ \& \ g_2 \ \& \ \neg g_3 \ (g_1 \ P_1' \parallel P_2' \parallel g_3 \ P_3') \\ [] \ \neg g_1 \ \& \ \neg g_2 \ \& \ g_3 \ (g_1 \ P_1' \parallel g_2 \ P_2' \parallel P_3') \end{pmatrix}$$

8. from 7, can we generalize for $P = P_1 \| ... \| P_n$? The answer is not easy.

   Here, we try to explain it through an implementation.

   We have n orthogonal sub-states and $P_i = g_i \; P'_i, \; i = 1..n$.

   Let $S = \{S_0, ..., S_{2^n-2}\}$ is set of binary representations of $\{0, ..., 2^n - 2\}$.

   For example, with $n = 3$:

   $S_0 = 000$, $S_1 = 001$, ..., $S_6 = 110$

   and we call

$$G_{ki} = \begin{cases} g_i & S_{ki} = 0 \\ \neg g_i & S_{ki} = 1 \end{cases} \quad k = 0..2^n - 2, i = 1..n$$

$$Q_{ki} = \begin{cases} P'_i & S_{ki} = 0 \\ g_i \; P'_i & S_{ki} = 1 \end{cases} \quad k = 0..2^n - 2, i = 1..n$$

   then we have,

$$P = \Big(\; []_{k=0}^{2^n-1} \; G_{k1} \; \& \; ... \; \& \; G_{kn} \; (Q_{k1} \; \| \; ... \; \| \; Q_{kn}) \Big)$$

2.3 Summary of the notation used in this thesis

In this section we summarise some notations used in this thesis.

In short:

$\|_{1 \le i \le n} \; P_i$: is a short representation of $P_1 \; \| \; ... \| \; P_n$

$[]_{1 \le i \le n} \; P_i$: is a short representation of $P_1 \; [] \; ... [] \; P_n$

$\&_{1 \le i \le n} \; h_i$: is a short representation of $h_1 \; \& \; ... \& \; h_n$

If sc is a statechart and T is composed of all possible transition τ among immediate sub-states of sc, we have:

$active(sc)$: is active sub–state of sc.

$resc(\tau, sc)$: is a function that returns a statechart which is the result of sc after transition τ is fired. The type of $resc$ is:

$$resc : \mathcal{T} \times \mathtt{SC} \rightarrow \mathtt{SC}$$

$T^*(sc)$: is a set contains all possible transitions inside `Or`-statechart sc along its transitive active sub-state chain, like: $T^*(sc) =_{df} \{\tau \mid \tau \in T \wedge src(\tau) = p_l\} \bigcup T^*(p_l)$.

or-depth(sc) : is used to calculate the *or-depth* of sc.

If τ is a transition, then

$src(\tau)$: returns a source state of τ.

$tgt(\tau)$: returns a target state of τ.

Chapter 3

Mapping algorithm

In this chapter we present the mapping algorithm from statecharts to abstract Verilog. We also use some simple examples to illustrate the algorithm. After that we discuss some related works.

3.1 Mapping algorithm

We apply the algorithm that takes statecharts as input and outputs the abstract Verilog code which was defined in chapter 2. This mapping algorithm works in a top–down manner starting from the root of the statechart and then moving to its children. Each time, we consider the input statechart (each part of Statecharts) as a singleton statechart and continue until no further applicable.

We present the mapping function L as originally proposed in [37] which deal with each type of source statechart. It means that the algorithm divides the input statechart into three kinds; `Basic`, `Or`, and `And`-statechart, then constructs the output with each case. The function is defined below:

Definition of mapping function L:

$$L : \texttt{SC} \rightarrow \texttt{Verilog}$$

maps any statechart description into a corresponding Verilog process. It keeps unchanged the set of variables employed by the source description, i.e.,

$$\forall sc \in \texttt{SC} \bullet \mathbf{vars}(L(sc)) = \mathbf{vars}(sc)$$

and it is inductively defined as follows.

- For a statechart $sc = |[s]|$ constructed by `Basic`, L maps its input into an idle program *sink* which can do nothing but let time advance, i.e.,

  $$L(sc) =_{df} sink$$

- For a statechart $sc = |[s : \{p_1, ..., p_n\}]|$ constructed by `And`, L maps its input into a parallel construct in Verilog.

  $$L(sc) =_{df} \|_{1 \leq i \leq n} L(p_i)$$

- For a statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$ constructed by `Or`, we define L by exhaustively figuring out the first possible transitions of sc if any, otherwise it returns *sink*.

  $$L(sc) =_{df} \begin{cases} sink & \text{if } T^*(sc) = \emptyset \\ P & \text{otherwise} \end{cases}$$

  where

  $$\begin{aligned} P \;\; =_{df} \;\; & []_{0 \leq k \leq or-depth(sc)} \; [] \{ b_{\tau_k} \;\&\; g^i_{\tau_k} \;\&\; (\&_{0 \leq j \leq k} \, h_j) \;\&\; g^0_{\tau_k} \; L(resc(\tau_k, sc)) \;\; | \\ & \tau_k \in T(active^k(sc)) \;\; \wedge \;\; src(\tau_k) = active^{k+1}(sc) \; \wedge \\ & h_j = \&\{\neg g^i_\tau \mid \tau \in T(active^{j-1}(sc)) \;\; \wedge \;\; src(\tau) = active^j(sc)\}\} \end{aligned}$$

  and

$$\begin{array}{lll} active^0(sc) & =_{df} & sc \\ active^1(sc) & =_{df} & active(sc) \\ active^{i+1}(sc) & =_{df} & active(active^i(sc)) \end{array}$$

For each statechart, we always assume each of its variables has bounded range, and the set of possible events is finite, which implies that the set of its configurations is finite. Therefore, the set of configurations (under transition relation) forms a well–founded quasi order, which indicates the mapping function L is terminating.

Following are some formal notations used in the above definition. Firstly, the function $or-depth$: SC $\rightarrow N$ to calculate the "or–dept" of a statechart, which is defined as follows:

- for a statechart $sc = |[s]|$ constructed by `Basic`, $or\text{–}depth(sc) =_{df} 0$;
- for a statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$ constructed by `Or`, $or-depth(sc) =_{df} or-depth(p_l) + 1$;
- for a statechart $sc = |[s : \{p_1, ..., p_n\}]|$ constructed by `And`, $or-depth(sc) =_{df} 1$.

The *or–depth* of an `Or`-chart just records the depth of the path transitively along its active `Or`-sub-states. We stop going further once an `And`-state is encountered. The *or–depth* of an `And`-chart is simply 1.

Secondly, the source and target state functions, $src(\tau)$ and $tgt(\tau)$, respectively represent the source and target state of a transition τ. Given a transition $\tau = \&_{1\leq k\leq m}\tau_{i_k} \in T$, where $\tau_{i_k} \in T^*(p_{i_k})$, for $1 \leq k \leq m$, and $i_1, ..., i_n$ is a permutation of $1, ..., n$, we define its source and target state as follow:

$src(\tau) =_{df} (q_1, ..., q_n)$, where $q_{i_k} = src(\tau_{i_k})$, for $1 \leq k \leq m$, and $q_{i_k} = active(p_{i_k})$, for $m < k \leq n$;

$tgt(\tau) =_{df} (r_1, ..., r_n)$, where $r_{i_k} = tgt(\tau_{i_k})$, for $1 \leq k \leq m$, and $r_{i_k} = active(p_{i_k})$, for $m < k \leq n$.

Where, $T^*(p)$ contains all possible transitions inside p along its transitive active sub-state chain, i.e., $T^*(p) =_{df} \{\tau \mid \tau \in T \wedge src(\tau) = p_l\} \cup T^*(p_l)$. And $active(sc)$ denotes a current active sub-state of sc. With an `Or`-statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$, we have $active(sc) = p_l$. With an `And`-statechart $sc = |[s : \{p_1, ..., p_n\}]|$, we have the active state is a vector of the active states of these constituents, i.e., $active(sc) =_{df} (active(p_1), ..., active(p_n))$.

Thirdly, we need to know the resulting statechart after a transition is taken. When a transition τ occurs, any involved statechart can have changes in its (transitive) active sub-states. We use a function:

$resc : \mathcal{T} \times \texttt{SC} \rightarrow \texttt{SC}$

to return the modified statechart after performing a transition in a statechart. It is defined inductively with regard to the type of the statechart.

- for a `Basic`-statechart sc, and any transition τ, $resc(\tau, sc) =_{df} sc$;
- for an `Or`-statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$, and a transition τ,

$$resc(\tau, sc) =_{df} \begin{cases} sc_{[l \mapsto a2d(tgt(\tau))]}, \text{if } \tau \in T \wedge src(\tau) = p_l; \\ sc_{[l \mapsto resc(\tau, p_l)]}, \text{if } \tau \in T^*(p_l); \\ sc, \text{otherwise.} \end{cases}$$

- for an `And`-statechart $sc = |[s : \{p_1, ..., p_n\}]|$, and a transition τ,

$$resc(\tau, sc) =_{df} \begin{cases} sc_\tau, \text{if } \tau = \&_{1 \leq k \leq m} \tau_{i_k} \in T(sc); \\ sc, \text{otherwise.} \end{cases}$$

where $sc_\tau = sc[q_1/p_1, ..., q_n/p_n]$ is the statechart obtained from sc via replacing p_i by q_i, for $1 \leq i \leq n$, $q_{i_k} = resc(\tau_{i_k}, p_{i_k})$, for $1 \leq k \leq m$, and $q_{i_k} = p_{i_k}$, for $m < k \leq n$.

The function $a2d(sc)$ is used to change the active sub-state of sc into its default sub-state, and the same change is applied to its new active sub-state. This function is defined as:

- $a2d(|[s]|) =_{df} |[s]|$
- $a2d(|[s : [p_1, ..., p_n], p_l, T]|) =_{df} |[s : [p_1, ..., p_n], a2d(p_1), T]|$
- $a2d(|[s : \{p_1, ..., p_n\}]|) =_{df} |[s : \{a2d(p_1), ..., a2d(p_n)\}]|$

The substitution $sc_{[l \mapsto p_m]}$ for an `Or`-statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$ is defined by $sc_{[l \mapsto p_m]} =_{df} |[s : [p_1, ..., p_n], p_m, T]|$

We present some simple examples in section 3.2 to illustrate the mapping algorithm. There are some more complicated statecharts we discuss in case studies in chapter 5. The specifications of these case studies can be handled by our implementation.

Note that the result of function L is the abstract Verilog code which is based on guarded choices (not real Verilog code yet). We discuss as the key difference from concrete Verilog code in the next section.

3.2 Some simple examples

We provide two groups of examples here. The first group has two `Or`-statecharts and the next has two `And`-statecharts.

3.2.1 Example of `Or`-statecharts

First example is shown in Fig. 3.1.

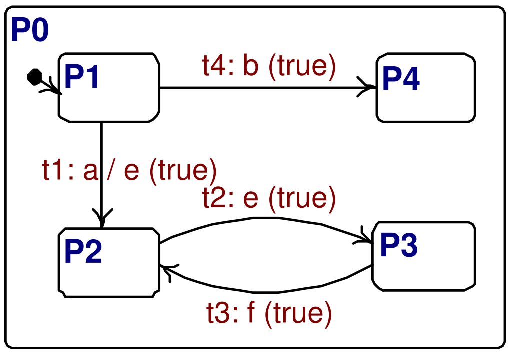


Figure 3.1: Example of simple statechart 1.

The textual specification of this example is:

States:

```
P0 = |[ S1: [ P1, P3, P2, P4 ], P1, { t2, t3, t1, t4 } ]|
P1 = |[ S2 ]|
P3 = |[ S3 ]|
P2 = |[ S4 ]|
P4 = |[ S5 ]|
```

Transitions:

```
t1 = < P1, { a }, { e }, true, P2 >
t2 = < P2, { e }, {  }, true, P3 >
t3 = < P3, { f }, {  }, true, P2 >
t4 = < P1, { b }, {  }, true, P4 >
```

After applying the mapping algorithm, we will obtain the abstract Verilog code:

$$L(ex1) \;=\; (\;(\;a\;\&\;\;\rightarrow e\;(\;fix\;X2.\;(\;e\;(\;f\;\;X2\;))))\;[]\;(\;b\;sink))$$

This example contains four transitions and there is one recursive process (from state P2 to P3 by t2 and back by t3) in the statechart. The recursive process is represented by $fix\ X2$. Process will $sink$ if transition t4 is taken and event b occurs when the control is in state P1.

Second example is more complicate, as shown in Fig. 3.2.

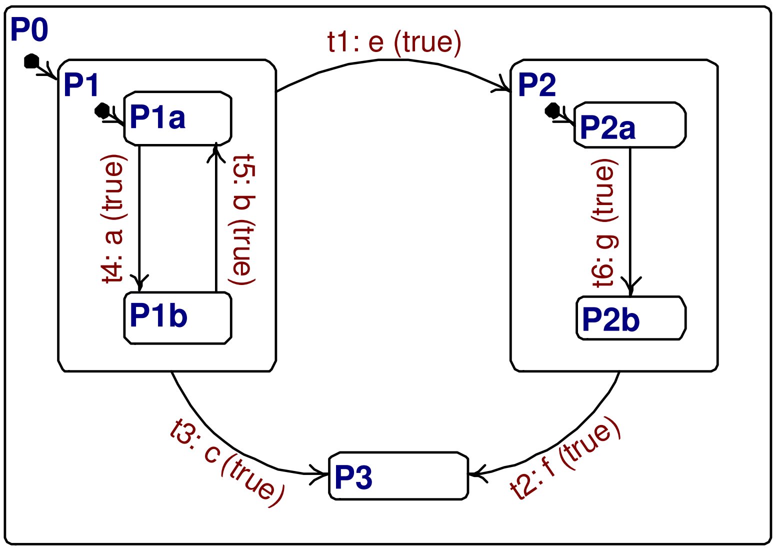


Figure 3.2: Example of simple statechart 2.

The specification of the example in Fig. 3.2 is:

States:

```
P0 = |[ S1: [ P1, P2, P3 ], P1, { t1, t2, t3 } ]|
P1 = |[ S2: [ P1a, P1b ], P1a, { t4, t5 } ]|
P2 = |[ S3: [ P2a, P2b ], P2a, { t6 } ]|
P3 = |[ S4 ]|
P1a = |[ S5 ]|
P1b = |[ S6 ]|
P2a = |[ S7 ]|
P2b = |[ S8 ]|
```

Transitions:

```
t1 = < P1, { e }, {  }, true, P2 >
t2 = < P2, { f }, {  }, true, P3 >
t3 = < P1, { c }, {  }, true, P3 >
t4 = < P1a, { a }, {  }, true, P1b >
t5 = < P1b, { b }, {  }, true, P1a >
t6 = < P2a, { g }, {  }, true, P2b >
```

After applying the mapping algorithm, we get the following abstract Verilog process:

$$L(ex2) \;=\; fix\; X1.(\; P1 \;[]\; P1a)$$

where:

$$P1 \;=\; ((\; e\; (\; P2 \;[]\; P2a))\; []\; (\; c\; sink))$$

$$P2 \;=\; (\; f\; sink)$$

$$P1a \;=\; ((\; a \;\&\; \neg c \;\&\; \neg e)\; (\; b \;\&\; \neg c \;\&\; \neg e)\; X1)$$

$$P2a \;=\; (\; g \;\&\; \neg f\; P2)$$

and Px is process start from state `Px`, like $P1a$ is process start from state P1a.

We can see that when the process goes to state P1b after transition $t4$ is taken. However, the process can still continue with transition $t1$ or $t3$ if event e or c occurs. This is because the transition of parent state has higher priority than that of its children.

3.2.2 Example of `And`-statecharts

We will now present an `And`-statechart with two children, as illustrated in Fig 3.3:

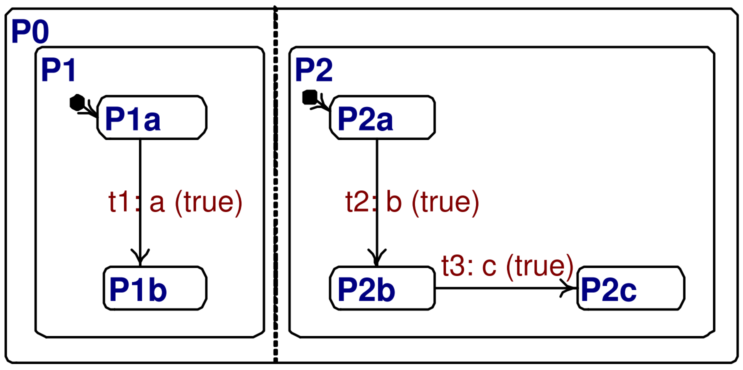


Figure 3.3: Example of simple statechart 3.

The textual specification is:

States:

```
P0 = |[ S1: { P1, P2 } ]|
P1 = |[ S2: [ P1a, P1b ], P1a, { t1 } ]|
P2 = |[ S3: [ P2a, P2b, P2c ], P2a, { t2, t3 } ]|
P1a = |[ S4 ]|
P1b = |[ S5 ]|
P2a = |[ S6 ]|
P2b = |[ S7 ]|
P2c = |[ S8 ]|
```

Transitions:

```
t1 = < P1a, { a }, {  }, true, P1b >
t2 = < P2a, { b }, {  }, true, P2b >
t3 = < P2b, { c }, {  }, true, P2c >
```

After applying the mapping algorithm, we obtain the following abstract Verilog process:

$$L(ex3) \;=\; P1a \parallel P2a$$

where:

$$P1a = (\ a\ sink)$$

$$P2a = (\ b\ c\ sink)$$

The last example is illustrated in Fig. 3.4.

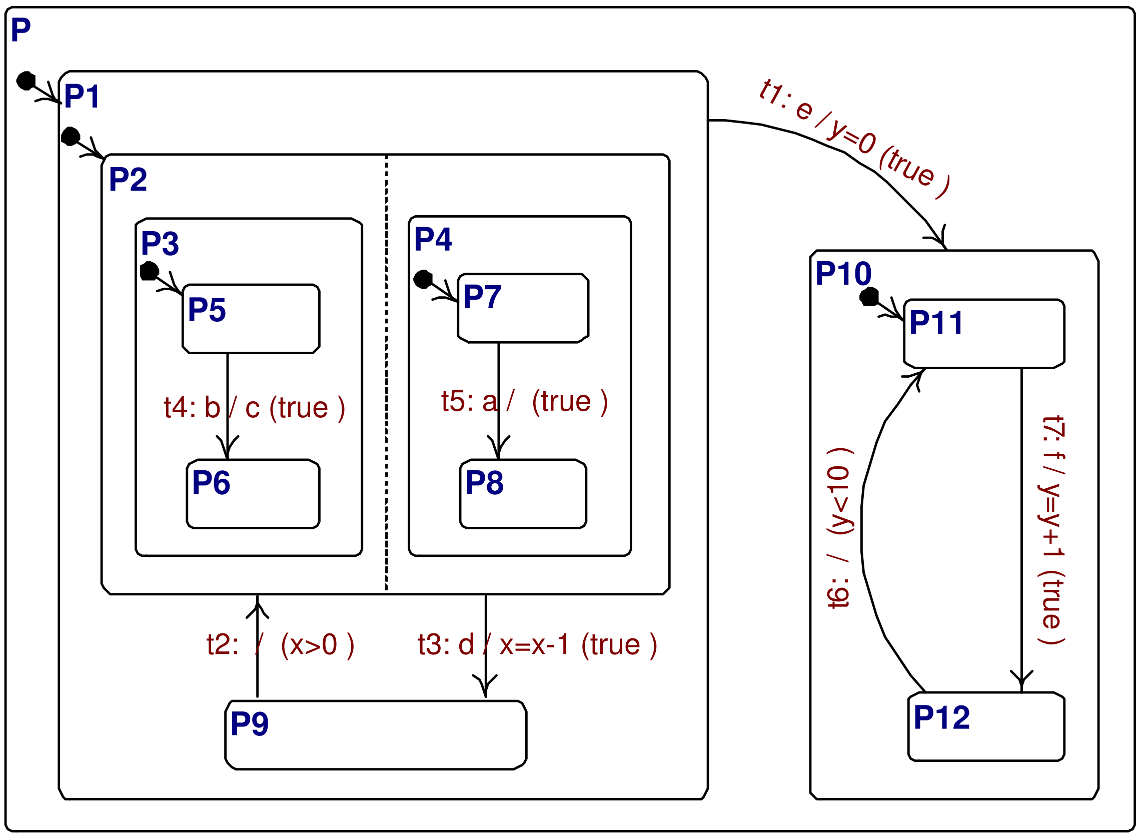


Figure 3.4: Example of simple statechart 4.

The textual specification is:

States:

```
P = |[ S1: [ P1, P10 ], P1, { t1 } ]|
P1 = |[ S2: [ P2, P9 ], P2, { t3, t2 } ]|
P10 = |[ S3: [ P11, P12 ], P11, { t7, t6 } ]|
P2 = |[ S4: { P3, P4 } ]|
P9 = |[ S5 ]|
P3 = |[ S6: [ P5, P6 ], P5, { t4 } ]|
P4 = |[ S7: [ P7, P8 ], P7, { t5 } ]|
P5 = |[ S8 ]|
P6 = |[ S9 ]|
P7 = |[ S10 ]|
P8 = |[ S11 ]|
P11 = |[ S12 ]|
P12 = |[ S13 ]|
```

Transitions:

```
t1 = < P1, { e }, { y=0 }, true, P10 >
t2 = < P9, {  }, {  }, x>0, P2 >
t3 = < P2, { d }, { x=x-1 }, true, P9 >
t4 = < P5, { b }, { c }, true, P6 >
t5 = < P7, { a }, {  }, true, P8 >
t6 = < P12, {  }, {  }, y<10, P11 >
t7 = < P11, { f }, { y=y+1 }, true, P12 >
```

After applying the mapping algorithm, we obtain the following abstract Verilog process:

$$L(ex4) \ = \ fix\ X1.(\ P1 \ [] \ P2 \ [] \ (\ g0 \ [] \ g1 \ [] \ g2))$$

where:

$$P1 \ = \ ((\ e \ \& \ @(y=0)\ )\ P11)$$

$$P2 \ = \ ((\ d \ \& \ \neg e \ \& \ @(x=x-1))\ (\ \neg e \ \& \ (x>0)\ X1))$$

$$P5 \ = \ ((\ b \ \& \ \neg e \ \& \ \neg d \ \& \ \rightarrow c)\ P8)$$

$$P7 \ = \ (\ a \ \& \ \neg e \ \& \ \neg d\ P8)$$

$$P8 \ = \ P2 \ [] \ P1$$

$$P11 \ = \ (\ fix\ X12.((\ f \ \& \ @(y=y+1))\ (\ \& \ (y<10)\ X12)))$$

$$g0 \ = \ (\ b \ \& \ \neg a \ \& \ \neg e \ \& \ \neg d \ \& \ \rightarrow c)\ (\ P2 \ [] \ P1 \ [] \ P7)$$

$$g1 \ = \ (\ a \ \& \ \neg b \ \& \ \neg e \ \& \ \neg d)\ (\ P2 \ [] \ P1 \ [] \ P5)$$

$$g2 \ = \ ((\ b \ \& \ a \ \& \ \neg e \ \& \ \neg d \ \& \ \rightarrow c)\ P8)$$

Above are examples of four statecharts, these examples are used to illustrate the mapping algorithm and their results. In chapter 5 we shall discuss two more complicated examples.

3.3 The replacement of guarded choices

The mapping algorithm presented above is to map a statechart specification into a program in abstract Verilog. However, we are expecting to do one step further; we are trying to get a concrete Verilog code form abstract version by eliminating all guarded choices. For example if we have:

$L(sc) \ = \ g1\ P1\ []\ g2\ P2$

then we can replace it by a parallel composition like $L(sc) \ = \ g1\ P'1 \parallel g2\ P'2$.

This is a very simple example so we can easy to understand. However, we have to figure out the common way for bigger process of abstract Verilog. This is more difficult and we are now still working on it.

3.4 Related work

There are many works related to translate or to map statecharts into other language [1, 4, 5, 9, 29, 40, 41, 42, 43, 44, 47]. Beauvais et.al. [4] presented their study on translating statecharts into *Signal* language. Their aim is to use *Signal* (a synchronous language) and its environment for formal verification purposes. Other supporting tool for formal verification purposes is provided by Seshia et.al. [43]. This work translates statecharts to Esterel language. However, both of these translations are based on the informal semantics of statechart from [15]. There is no guarantee of the correctness. Other attempted works are David [9] and Mikk [29], the authors discuss the translation of statecharts into another graphical formalism called *Extended Hierarchical Automata* (EHA) and used UPPAAL/SPIN tools to check properties of statechart models. In this formulation, the inter–level transitions are eliminated, by extending the label to include *source_restriction* and *target_determinator* sets. From the denotation semantics of [17], Sowmya et.al. [44] state their aim to connect a subset of statecharts with temporal logic *FNLOG* for theoretically proving statecharts' properties. And then Almeida Júnior [27] employed this idea and developed an adaptive models for systems description.

A translation from statecharts to B/AMN is reported by Sekerinski et.al. [40, 41, 42]. This work takes hierarchical code generation approach and a tree of nested layers is encoded in a nested switch statement or in a class hierarchy with virtual methods (see [1] for general version). The statechart semantic in [40] is based on [29] and no correctness issue has been given.

Two other works on efficient code synthesis from statechart are Björklund et.al. [5] and Wasowski [47]. Björklund's work devises an intermediate language to gain the efficiency, but the use of flattening in course of translation may cause code explosion. Wasowski presents a technique to represent statecharts as a hierarchy tree and use two arrays to store the tree.

Chapter 4

Implementation

4.1 Overview

Our implementation consists of two parts: a statechart editor and a mapping program from statechart into abstract Verilog.

- The first part, called Statechart_E, is a stencil that was built as an add-on of Microsoft Visio 2002.
- The second part, called AMSV (Automatic Mapping of Statechart into Verilog), is essentially a Java program.

Fig. 4.1 shows the stages of using our system. Users first draw their statecharts, using Statechart_E, which also automatically generates the corresponding textual representations. AMSV will then generate abstract Verilog code from textual representation of these statecharts. In next two sections, we will discuss about Statechart_E, AMSV, and some other techniques used in the system.

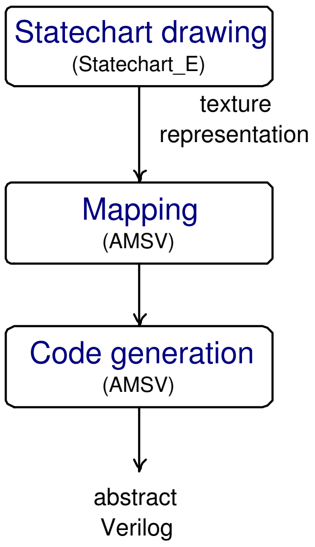


Figure 4.1: Structure of the implementation.

4.2 Statechart editor

Our statechart editor is built with three main purposes:

- First, of course is for editing Statechart diagrams. The editor should be convenient to use and easy to draw.
- Second, it should also be easy to export textual representation of statechart. This is used by the mapping algorithm which converts statechart to abstract Verilog.
- Last, it should be easy to save the statecharts to other graphical formats (like bmp, jpg, ps, eps, etc) This is important for portability and for documentation.

From these requirements, we built Statechart_E as an add-on/embedded stencil in Microsoft Visio. We make use of MS. Visio because Visio is a very powerful graphical editor tool for drawing diagrams. Visio also supports many graphical formats for exporting our diagrams. Moreover, using Visio, we can not only draw statechart components but also other shapes from suitable drawing types or stencils.

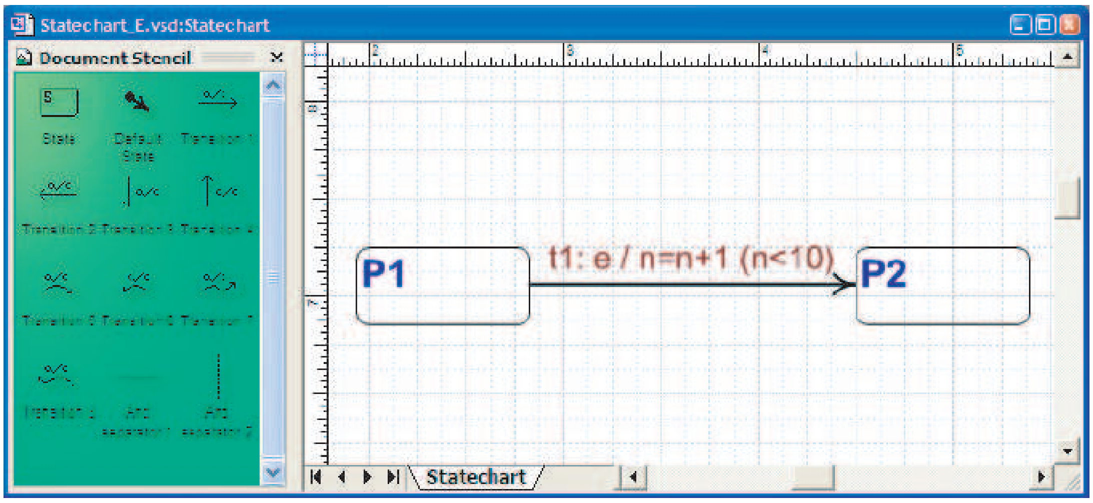


Figure 4.2: Statechart_E interface.

Fig. 4.2 shows how Statechart_E stencil looks like. The left hand side is a group of masters to draw statechart components. The right hand side is a statechart diagram under construction.

Features of Statechart_E:

- A menu named *Statechart* is added to the menu bar of Visio as illustrated in Fig. 4.3. This menu contains two functions, namely: *Generate statechart* and *Add new statechart page*. The first function is used to export the current statechart to a textual file. This file is used as input for the mapping program which to transform to abstract Verilog. The second function is used to add a new page for current statechart diagram. To enable this menu and its functions, users must allow a macro to be accepted when opening the stencil.

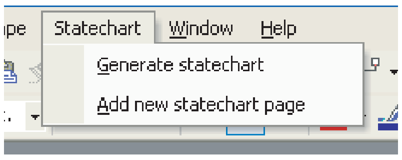


Figure 4.3: Statechart_E menu.

- A set of masters is added to the stencil and this is used for constructing statecharts. It consists of a state master, a default master (common for all kind of states), 8 transition masters (to help build complex statecharts), and vertical/horizontal separators for And-state. The left hand side of Fig. 4.2 shows these masters.

- Each master is accompanied by a program written in Visual Basic for Application (VBA) to check data, events and perform actions of each master. Some masters are linked to a window to allow input of needed data. This program also partially checks the supplied data such as duplicate name, etc.

- We also allow users to build hierarchical statecharts. Users can easily extend a given statechart by adding a new page (using the second function in menu *Statechart*) and continue to extend the current statechart in a hierarchical manner in the new page. For example, a user may draw the sub-states of P1 and P2 of Fig. 4.2 using two new pages. Note that *generate* function will read all components in all pages of the statechart.

How to use Statechart_E:

To draw a new statechart, users need to first open stencil and enable its macro. The usage of Statechart_E stencil and its masters is almost the same as other stencils in Visio. However, the generation function does not work with external components. Chapter 5 will present some bigger examples when we describe some case studies.

4.3 AMSV - Core mapping program

4.3.1 DFS algorithm

As presented in chapter 3, the mapping algorithm has to deal with each state; `Basic`, `And`, and `Or` states. It can construct the corresponding Verilog code after the mapping algorithm has been applied to all states of the source statechart. Nevertheless, how do we traverse all states of the input statechart? In the AMSV, we make use of depth–first–search (DFS) algorithm [8] to reach all states of the statechart.

However, DFS works on each tree of nodes. To apply DFS we have to reconstruct the source statechart into a tree of states. Fig. 4.4 shows an example of hierarchy tree (b) for a simple statechart (a). Here, dashed arrows denote the children of an `And`-state (like arrow from P0 to P1, P2), while the doted arrows point to the active sub-states of `Or`-state (like arrow from P1 to P3 or P2 to P6). The solid arrows represent the transitions.

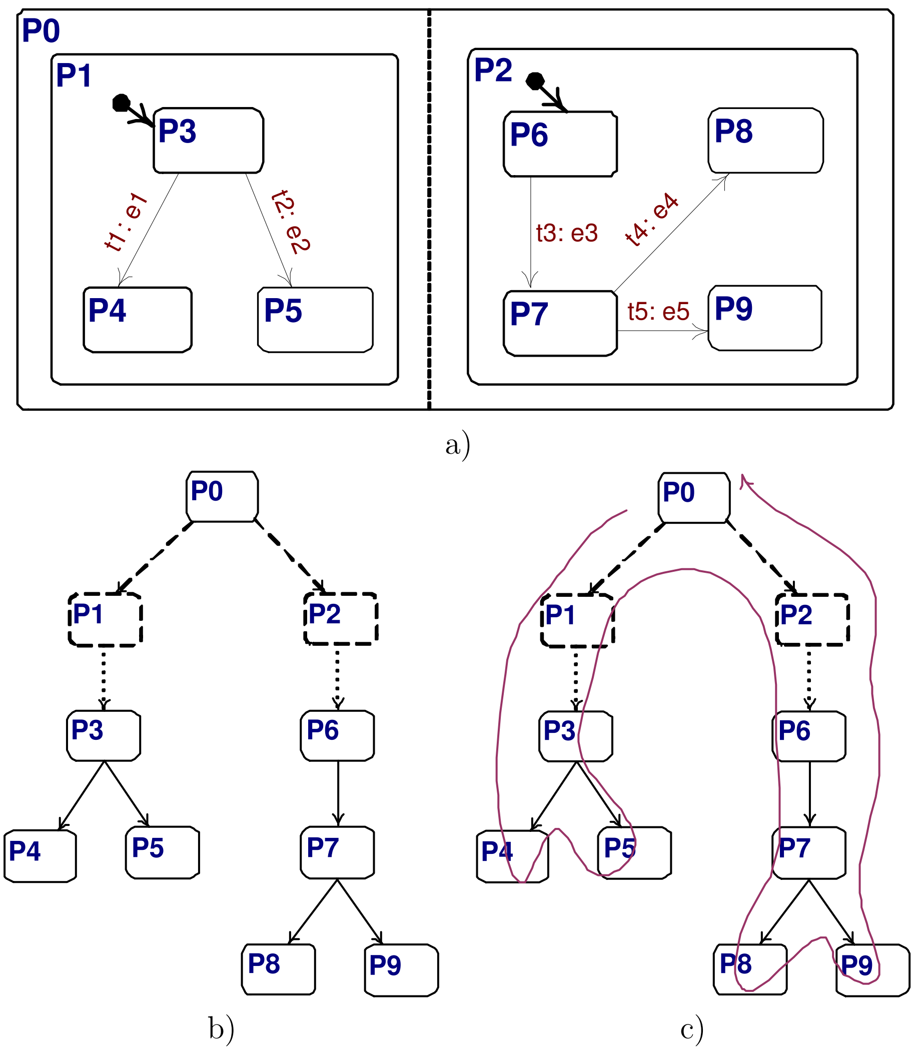


Figure 4.4: Hierarchy tree. a) Statechart example, b) hierarchy tree, and c) DFS route.

After reconstructing each statechart into a hierarchy tree, we apply a recursive function which maps each statechart to abstract Verilog. At each time, we only consider one state, called the current state. Through this recursive function, we apply the mapping algorithm to all states of the source statechart to obtain Verilog process code. These codes are kept in a hash table for latter use. After that, we gather the output code (from sub-states or from target states of all transitions to the current state) to generate final abstract Verilog process.

For example, in the Fig. 4.4, first we start from the root state (like P0). After that, we invoke the function itself if it is possible to go to current state's children (P1, P2) or target states of transitions (P3 to P4, P5). A systematic way of finding the next state is described below. Fig. 4.4 c shows the route taken by our DFS traversal:

- each state is the target of transition: If there exists any transition from the current state, go to the target state of the transition. Like transitions from P3 to P4 or P5. The information of the transition will be memorized to generate output code. If there are more than one transitions from current state, process it one by one. The order between these transitions is not important.

- each state is a child of the `And`-state: If the current state is `And`-state, go to all children. Like from P0 to P1 or P2. Information of children in that `And`-state will be memorized during code generation, as acquired by the Verilog language.

- state is sub-state of `Or`-state: Just go to active state and continue as before. For example, P3 and P6 are the active states of P1 and P2.

4.3.2 Recursion

During the traversal to the states of a given statechart, it is possible for a transition to re-occur. This may be due to non-termination. To solve this problem we use a boolean array to remember all states which the program has already encountered. If a program reaches a marked state, it just uses that information to generate a loop, and then go back to previous state. This is meant to terminate a recursive transition.

4.3.3 Parallel expansion

Recall from section 2.2.3, we have to take into account the parallel expansion of `And`-state. Whenever an `And`-state is reached, all information (guards, conditions, etc) of the children of a current state are used for expansion. The only exception is when the current state is the root. In this case we generate Verilog code from all its children and gather it using the parallel operation (||). This situation was discussed in section 2.2, with Fig. 3.3 and Fig. 3.4 as examples.

4.3.4 AMSV Program Structure

AMSV was built using the Java programming language. Here are brief technical specifications of AMSV:

- AMSV is written in Java SDK 1.4.1.
- AMSV reads a statechart specification (list of states and transitions) from

an input file (specify by argument) in textual specified format. The correctness of the input data is assumed to have been checked.

- AMSV transforms the source statechart into abstract Verilog and prints out the generated code as a text file.

AMSV program structure:

- The main class is `amsv.java`, for controlling the program.
- Two data structure classes (`State.java` and `Tranition.java`) are used to capture states and transitions.
- A class of data (`data.java`) is used to keep all information during the execution time.
- A core class (`mapping.java`) is used for the mapping algorithm.
- Two classes (`getInput.java` and `writetoFile.java`) are used to read and to create data from each input file and to write the result to output file.

4.4 Related work

There are several works and related software, like *Rhapsody*, *AnyStates*. However, these are expensive commercial products. There are some free graphical tools to edit statechart, like *Diagen*, *DOME*. Some of these tools have both GUI and code generation. We shall describe these tools briefly in what follows.

Some works are very old and no longer supported, for example, of Paulisch [48] and Lucas [49]. Their idea is to create a graphical interface to edit concurrent, hierarchical, finite state machines (CHSMs). Both systems are written in C++ in X–Windows environment. Another work almost at the same time is the work of Edwards [50] based on `tcl` package. These old tools are typically unable to handle larger statecharts.

Three more free tools are *Diagen* [51], *DOME* [52], and *Jgraphpad* [53]. Diagen (The *Diagram Editor Generator*) is a system for easy development of powerful diagram editors. It consists of two main parts: A framework of Java classes that provides generic functionality for editing and analyzing diagrams and a generator program that can produce Java source code for most of the functionality that depends on the concrete diagram language. DOME (the *DOmain Modeling Environment*) is a meta-CASE system suitable for building object oriented software models (Coad-Yourdon OOA and UML, for example), and more importantly, for building original models. It includes a graphical front-end, and a powerful back-end language for generating code, analysis and documentation. JGraphpad is a powerful diagram editor for Swing that offers XML, drag and drop, zoom, automatic layout, print support, and much more. JGraphpad, can be used to create flow charts, maps, UML diagrams, and networks with thousands of nodes. JGraphpad is available with sourcecode, which may be used to develop new (commercial) applications.

Commercial software are typical costly, such as *Rhapsody* [55], Rhapsody reverses the traditional design process, allowing you to find problems as they occur, versus waiting until the very end when they are far more costly to correct. Another commercial tool is I–logix's product *Statemate.* This is a graphical modelling and simulation tool. Another product is that of XJ Technologies, called AnyStates^{TM} [57], for state analysis. This aims at developing software components based on statecharts (state machines). Some key feature of Anystates are: state-of-the-art graphical statechart editor, synchronous graphical and textual views on a statechart, and on-the-fly code generation. BetterState$^{®}$ [56] (product of Wind River) is a graphical programming tool based on Statecharts and Flowcharts. With graphical specification, automatic code generation, graphical debugging, and round–trip engineering, BetterState offers embedded system developer's significant benefits. This include simpler software development, reduced design iterations, and easier maintenance and design reuse. Lastly, is Stateflow [58] (product of The MathWorks) is an interactive design tool for modelling and simulating event–driven systems. Tightly integrated with Simulink and MATLAB, Stateflow provides an elegant solution for designing embedded systems that contain supervisory logic. Its combination of graphical modelling and animated simulation brings system specification and design closer together.

Chapter 5

Case studies

In this chapter, we illustrate the mapping algorithm via the following examples: a CD player and a washing machine.

5.1 CD-player

5.1.1 Specification

Fig. 5.1 shows the graphical statechart of a CD-player. It contains two orthogonal regions: *Play control* (`PlayCtr`) and *Track information* (`TrackCtr`), which are used to control the playing mode and record the track information respectively. The first region contains `Stop, Play, Pause` sub-states to control the playing mode, while the second one contains only a sub-state, `Track`. Three buttons, `Next`, `Prev`, and `select a track`, are associated with the `Track` state. The variable ct (that is, current track) is used to keep record of the current position of the CD being played. We assume *ct* is initially 0 whenever the CD-player is switched on.

In this model, `Stop` and `Track` are respectively two default sub-states of two orthogonal regions. So when the CD-Player is switched on, both of them are entered simultaneously. Upon the arrival of event *Play_pressed* (that is, the `Play` button is pressed), transition $t1$ is taken and state `PlayingCtr` is entered, where the default sub-state `Playing` becomes active. Transitions $t4$ and $t3$ are used to alter between state `Playing` and `Paused`. Transition t2 connects state `PlayingCtr` with state `Stop`. When the control is in state `PlayingCtr` (either `Playing` or `Paused`), and $t2$ is enabled, it will yield the `Stop` state (that is, the CD-player will stop).

In the orthogonal state `TrackCtr`, upon the arrival of events *Next_pressed* or *Prev_pressed*, the variable *ct* (current track) will be changed according to the event. Conditions ($ct > 1$) and ($ct < Max(track)$) are used to check the range of the *ct*. The transition $t7$ is taken if users select any track in the range.

For simplicity, we only added track information in this specification of a CD-player. A real CD-player may contain other functionalities, like timer, forward, rewind, etc. We can add these setting as parallel regions in a similar way.

After drawing the statechart specification in Statechart_E, the following textual representation is automatically generated:

```
CD-Player-ON = |[ S1: { PlayCtr, TrackCtr } ]|
PlayCtr = |[ S2: [ Stop, PlayingCtr ], Stop, { t1, t2 } ]|
TrackCtr = |[ S3: [ Track ], Track, { t5, t7, t6 } ]|
Stop = |[ S4 ]|
PlayingCtr = |[ S5: [ Playing, Paused ], Playing, { t3, t4 } ]|
Playing = |[ S6 ]|
Paused = |[ S7 ]|
Track = |[ S8 ]|

t1 = < Stop, { Play_pressed }, { ct=1 }, true, PlayingCtr >
```

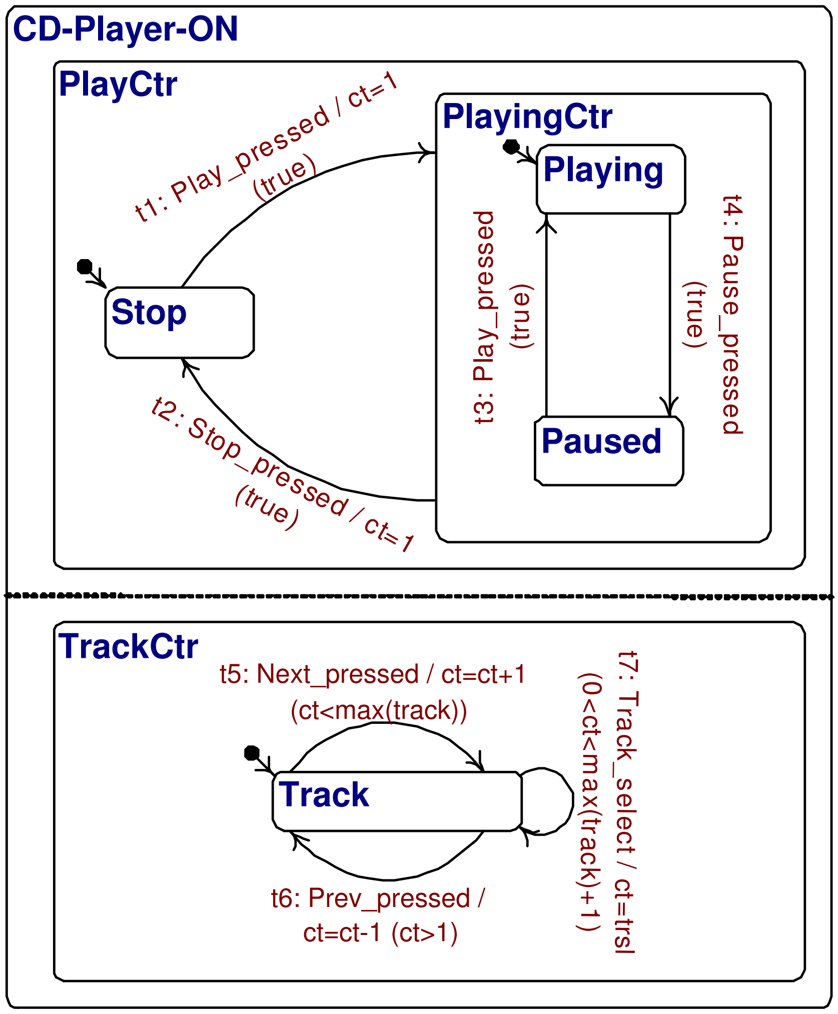


Figure 5.1: CD player with track information (ct).

```
t2 = < PlayingCtr, { Stop_pressed }, { ct=1 }, true, Stop >
t3 = < Paused, { Play_pressed }, {  }, true, Playing >
t4 = < Playing, { Pause_pressed }, {  }, true, Paused >
t5 = < Track, { Next_pressed }, { ct=ct+1 }, ct<max(track),
       Track >
t7 = < Track, { Track_select }, { ct=trsl }, 0<ct<max(track)+1,
       Track >
t6 = < Track, { Prev_pressed }, { ct=ct-1 }, ct>1, Track >
```

The first 8 lines are information of states. The rest are transitions.

5.1.2 Result

The textual representation given in last section is taken as the input of our algorithm AMSV, the output we obtain is the following code in abstract Verilog:

```
Result:
L_PlayCtr || L_TrackCtr

Where:
L_PlayCtr =  fix X0. ( L_Stop )
L_TrackCtr =  fix X2. (
  ( ( ( Next_pressed & @( ct=ct+1 ) & ( ct<max(track) ) X2 )
    [] ( Track_select & @( ct=trsl ) & ( 0<ct<max(track)+1 ) X2 ) )
  [] ( Prev_pressed & @( ct=ct-1 ) & ( ct>1 ) X2 ) ) )
L_Stop = ( ( Play_pressed & @( ct=1 ) )
  ( ( Stop_pressed & @( ct=1 ) X0 ) []  fix X1. ( L_Playing ) ) )
L_Playing = ( ( Pause_pressed &  not Stop_pressed )
              ( ( ( Play_pressed &  not Stop_pressed ) X1 )
                [] ( Stop_pressed & @( ct=1 ) X0 ) ) )
```

note that we use fix (rather than μ) to denote the recursion. *L_state* is the corresponding result from *state*.

Here we can see that the L_PlayCtrl and L_TrackCtr are processes which are running in parallel, where the recursive identifiers X0, X1, X2 represent three loop points.

5.2 Washing machine

5.2.1 Specification

In this subsection, we discuss a washing machine with five setting functions; `Timer`, `Hot water`, `Rinse level`, `Water level`, and `Pre-wash`. Fig. 5.2 shows the user interface of the washing machine. Fig. 5.3 gives the statechart specification of the washing machine corresponding to the interface, while Fig. 5.4 zooms into the sub-state `Washing-Ctr`. Statechart in Fig. 5.3 contains six parallel regions corresponding to five setting functions and the washing progress (*Wash-Ctr*). Each setting region contains a sub-statechart to change the value of its function. For example, in the `Timer-Ctr` region, the variable tm denotes the time that the washing machine has to wait before it starts to wash. It can be changed by `Inc` or `Dec` buttons. Other variables hw (hot water), rl (rinse level), wl (water level) and pw (pre-wash) are similar, and can be changed via pressing corresponding buttons. The default values of these variables are shown in Fig. 5.2 with black circles ($hw = 0$, $rl = 0$, $wl = 0$, and $pw = 0$) and default timer is 0.

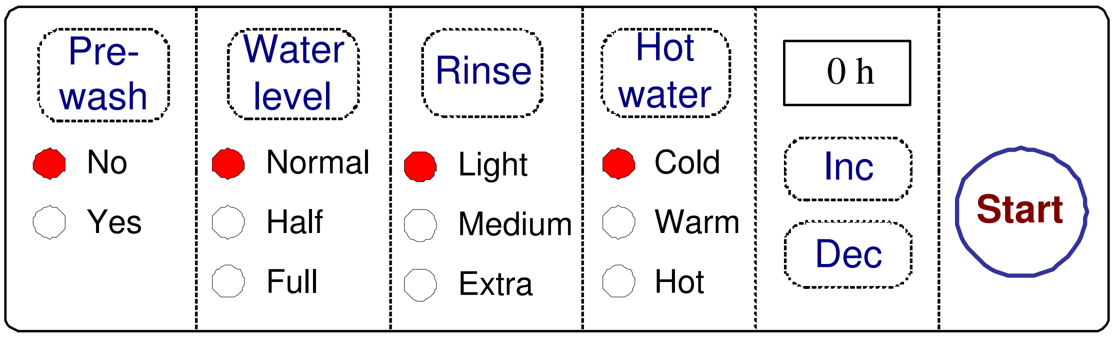


Figure 5.2: Interface of the washing machine.

The `Washing-Ctr` is an `Or`-state as given in Fig. 5.4. The state `Check-wait` is activated once state `Washing-Ctr` is entered. If tm is greater than 0, the machine keeps waiting for tm time before the control moves to `Pre-wash` state. The transition $t18$ calculates the value of the variable *washtime* based on the

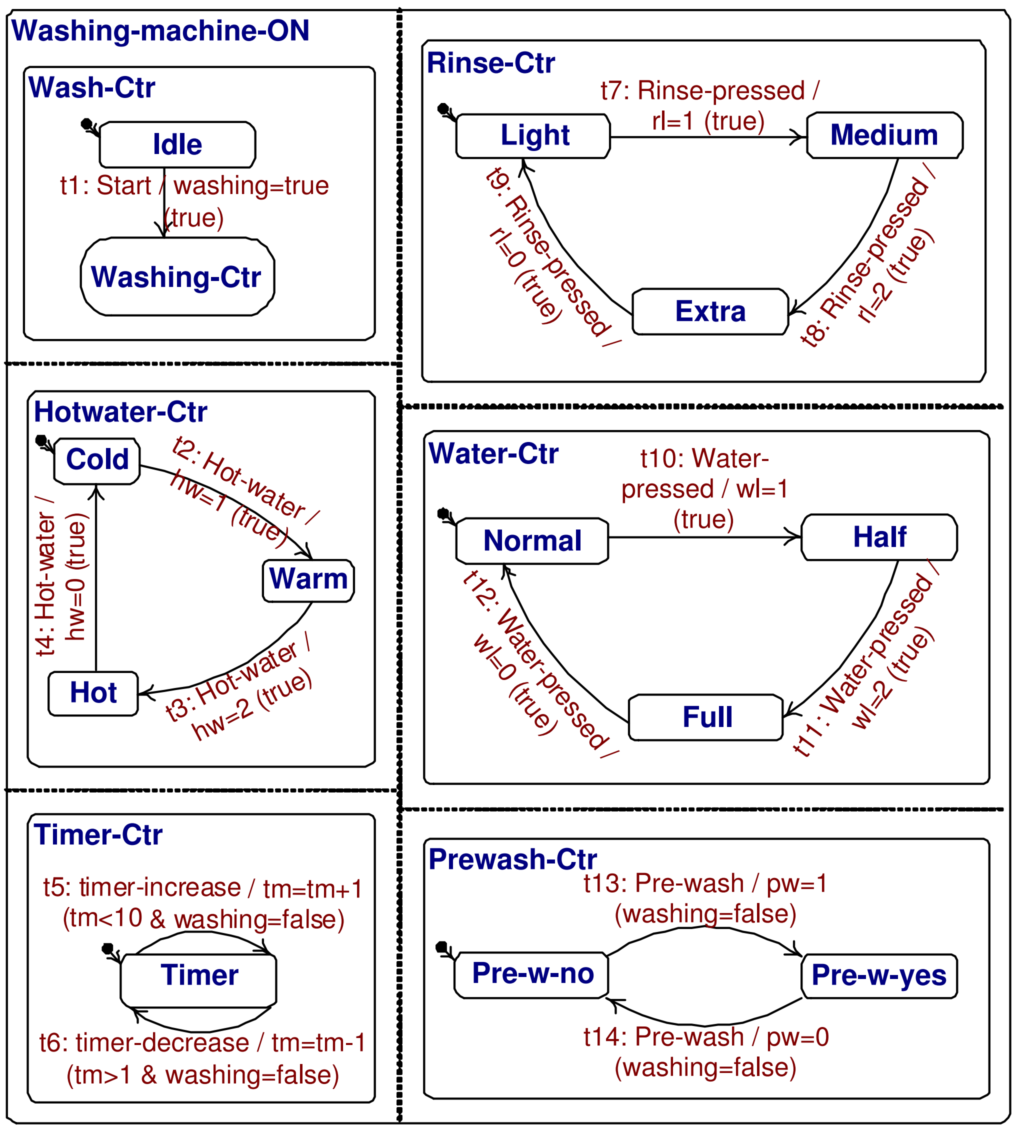

Washing-machine-ON
Wash-Ctr
Idle
t1: Start / washing=true (true)
Washing-Ctr
Rinse-Ctr
t7: Rinse-pressed / rl=1 (true)
Light
Medium
t9: Rinse-pressed / rl=0 (true)
Extra
t8: Rinse-pressed / rl=2 (true)
Hotwater-Ctr
Cold
t2: Hot-water / hw=1 (true)
Warm
t4: Hot-water / hw=0 (true)
Hot
t3: Hot-water / hw=2 (true)
Water-Ctr
t10: Water-pressed / wl=1 (true)
Normal
Half
t12: Water-pressed / wl=0 (true)
Full
t11: Water-pressed / wl=2 (true)
Timer-Ctr
t5: timer-increase / tm=tm+1 (tm<10 & washing=false)
Timer
t6: timer-decrease / tm=tm-1 (tm>1 & washing=false)
Prewash-Ctr
t13: Pre-wash / pw=1 (washing=false)
Pre-w-no
Pre-w-yes
t14: Pre-wash / pw=0 (washing=false)

Figure 5.3: Main statechart of a washing machine.

pre-wash setting. For example, if pw is 0 then $washtime = 1$. The variable $washtime$ is used to keep record of the time that the clothes have been washed so far. It is explained as follows:

- $washtime = 0$: if $pw = 1$, need pre-wash.
- $washtime = 1$: if $pw = 0$, no need pre-wash, need powder, no spin.
- $washtime = 2$ *or* 3: wash without powder, spin.

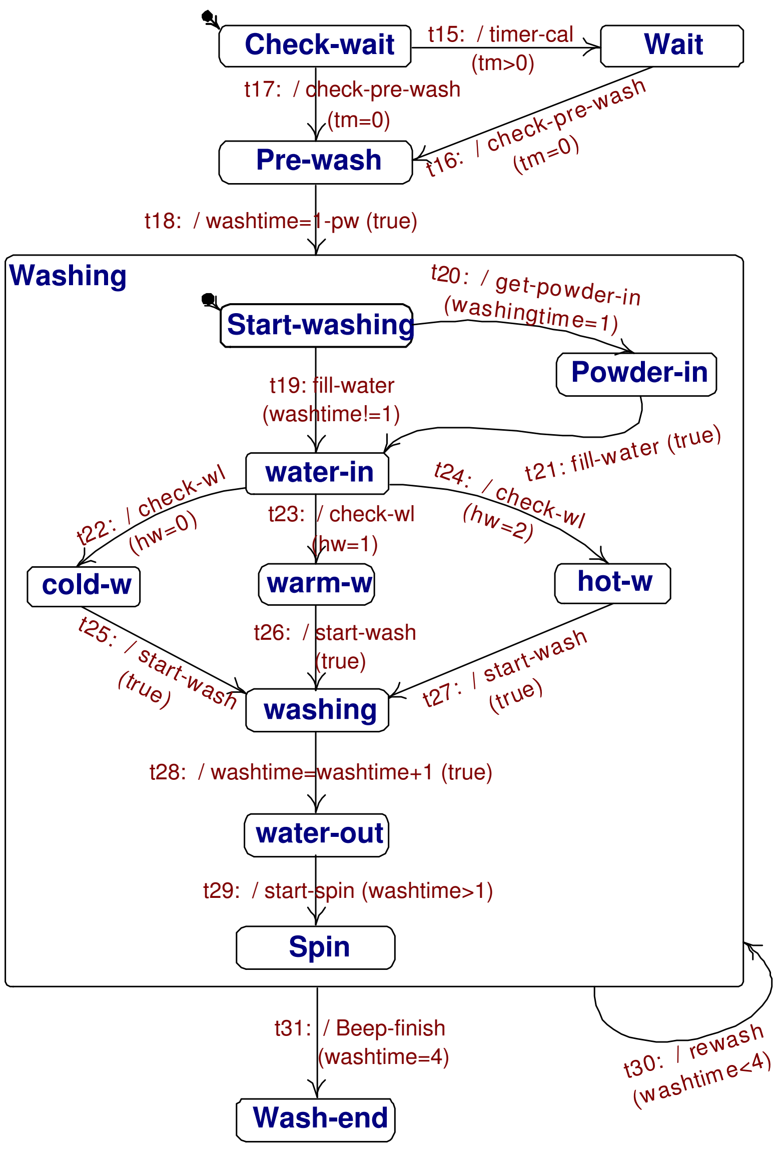


Figure 5.4: Statechart of `Washing-Ctr` in the washing machine.

- *washtime* > 3: finish.

Upon finishing, the machine beeps to inform the user.

The textual representation generated from Statechart_E is as follows:

```
Washing-machine-ON = |[ S1: { Wash-Ctr, Timer-Ctr, Water-Ctr,
                              Prewash-Ctr, Hotwater-Ctr, Rinse-Ctr } ]|
Wash-Ctr = |[ S2: [ Idle, Washing-Ctr ], Idle, { t1 } ]|
Idle = |[ S3 ]|
Washing-Ctr = |[ S4: [ Wait, Pre-wash, Washing, Wash-end, Check-wait ],
                 Check-wait, { t15, t16, t18, t17, t30 } ]|
Timer-Ctr = |[ S5: [ Timer ], Timer, { t5, t6 } ]|
Timer = |[ S6 ]|
Water-Ctr = |[ S7: [ Normal, Half, Full ], Normal, { t10, t11, t12 } ]|
Normal = |[ S8 ]|
Half = |[ S9 ]|
Full = |[ S10 ]|
Light = |[ S11 ]|
Medium = |[ S12 ]|
Extra = |[ S13 ]|
Prewash-Ctr = |[ S14: [ Pre-w-no, Pre-w-yes ], Pre-w-no,
                 { t13, t14 } ]|
Pre-w-no = |[ S15 ]|
Pre-w-yes = |[ S16 ]|
Hotwater-Ctr = |[ S17: [ Cold, Warm, Hot ], Cold, { t2, t3, t4 } ]|
Cold = |[ S18 ]|
Warm = |[ S19 ]|
Hot = |[ S20 ]|
Rinse-Ctr = |[ S21: [ Light, Medium, Extra ], Light, { t7, t8, t9 } ]|
Start-washing = |[ S22 ]|
Wait = |[ S23 ]|
Pre-wash = |[ S24 ]|
Washing = |[ S25: [ Start-washing, water-in, cold-w, warm-w, hot-w,
             washing, water-out, Powder-in, Spin ], Start-washing,
       t22, t24, t23, t25, t27, t26, t28, t29, t31, t19, t20, t21 } ]|
water-in = |[ S26 ]|
cold-w = |[ S27 ]|
warm-w = |[ S28 ]|
hot-w = |[ S29 ]|
washing = |[ S30 ]|
water-out = |[ S31 ]|
Powder-in = |[ S32 ]|
Spin = |[ S33 ]|
Wash-end = |[ S34 ]|
Check-wait = |[ S35 ]|

t1 = < Idle, { Start }, { washing=true }, true, Washing-Ctr >
t5 = < Timer, { timer-increase }, { tm=tm+1 }, tm<10 & washing=false,
       Timer >
```

```
t6 = < Timer, { timer-decrease }, { tm=tm-1 }, tm>1 & washing=false,
       Timer >
t10 = < Normal, { Water-pressed }, { wl=1 }, true, Half >
t11 = < Half, { Water-pressed }, { wl=2 }, true, Full >
t12 = < Full, { Water-pressed }, { wl=0 }, true, Normal >
t7 = < Light, { Rinse-pressed }, { rl=1 }, true, Medium >
t8 = < Medium, { Rinse-pressed }, { rl=2 }, true, Extra >
t9 = < Extra, { Rinse-pressed }, { rl=0 }, true, Light >
t13 = < Pre-w-no, { Pre-wash }, { pw=1 }, washing=false, Pre-w-yes >
t14 = < Pre-w-yes, { Pre-wash }, { pw=0 }, washing=false, Pre-w-no >
t2 = < Cold, { Hot-water }, { hw=1 }, true, Warm >
t3 = < Warm, { Hot-water }, { hw=2 }, true, Hot >
t4 = < Hot, { Hot-water }, { hw=0 }, true, Cold >
t15 = < Check-wait, {  }, { timer-cal }, tm>0, Wait >
t16 = < Wait, {  }, { check-pre-wash }, tm=0, Pre-wash >
t22 = < water-in, {  }, { check-wl }, hw=0, cold-w >
t24 = < water-in, {  }, { check-wl }, hw=2, hot-w >
t23 = < water-in, {  }, { check-wl }, hw=1, warm-w >
t25 = < cold-w, {  }, { start-wash }, true, washing >
t27 = < hot-w, {  }, { start-wash }, true, washing >
t26 = < warm-w, {  }, { start-wash }, true, washing >
t28 = < washing, {  }, { washtime=washtime+1 }, true, water-out >
t18 = < Pre-wash, {  }, { washtime=1-pw }, true, Washing >
t29 = < water-out, {  }, { start-spin }, washtime>1, Spin >
t31 = < Spin, {  }, { Beep-finish }, washtime=4, Wash-end >
t17 = < Check-wait, {  }, { check-pre-wash }, tm=0, Pre-wash >
t19 = < Start-washing, { fill-water }, {  }, washtime!=1, water-in >
t20 = < Start-washing, {  }, { get-powder-in }, washingtime=1,
        Powder-in >
t30 = < Washing, {  }, { rewash }, washtime<4, Washing >
t21 = < Powder-in, { fill-water }, {  }, true, water-in >
```

5.2.2 Result

We then run the AMSV algorithm to generate the Verilog program for the washing machine. We only give some part of the target code here. The full version of code can be found in Appendix B.2.

First of all, let us regard `Washing-Ctr` as a basic state (before we zoom into it).

We have the following Verilog program:

```
Result:
L_Wash-Ctr || L_Timer-Ctr || L_Water-Ctr || L_Prewash-Ctr ||
  L_Hotwater-Ctr || L_Rinse-Ctr

Where:
L_Wash-Ctr = L_Idle
L_Idle = ( Start & @( washing=true ) sink )
L_Timer-Ctr =
  fix X0. ( ( ( timer-increase & @( tm=tm+1 ) &
                ( tm<10 & washing=false ) X0 )
              [] ( timer-decrease & @( tm=tm-1 ) &
                   ( tm>1 & washing=false ) X0 ) ) )
L_Water-Ctr =  fix X1. ( L_Normal )
L_Normal = ( ( Water-pressed & @( wl=1 ) ) L_Half )
L_Half = ( ( Water-pressed & @( wl=2 ) )
           ( Water-pressed & @( wl=0 ) X1 ) )
L_Light = ( ( Rinse-pressed & @( rl=1 ) ) L_Medium )
L_Medium = ( ( Rinse-pressed & @( rl=2 ) )
             ( Rinse-pressed & @( rl=0 ) X4 ) )
L_Prewash-Ctr =  fix X2. ( L_Pre-w-no )
L_Pre-w-no = ( ( Pre-wash & @( pw=1 ) & ( washing=false ) )
               ( Pre-wash & @( pw=0 ) & ( washing=false ) X2 ) )
L_Hotwater-Ctr =  fix X3. ( L_Cold )
L_Cold = ( ( Hot-water & @( hw=1 ) ) L_Warm )
L_Warm = ( ( Hot-water & @( hw=2 ) ) ( Hot-water & @( hw=0 ) X3 ) )
L_Rinse-Ctr =  fix X4. ( L_Light )
```

The `sink` process in `L_Idle` is used to denote the `Washing-Ctrl` process, as we regard it as a basic state. On the other hand, if we consider `Washing-Ctr` as a stand-alone statechart, the corresponding code for it is as follows:

```
Result:
L_Check-wait =
  ( ( (  & @( timer-cal ) & ( tm>0 ) ) L_Wait )
    [] ( (  & @( check-pre-wash ) & ( tm=0 ) ) L_Pre-wash ) )
L_Start-washing =
  ( ( ( fill-water & ( washtime!=1 ) ) L_water-in
      ( & @( rewash ) & ( washtime<4 ) X0 ) )
    [] ( (  & @( get-powder-in ) & ( washingtime=1 ) ) L_Powder-in
         ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
L_Wait = ( (  & @( check-pre-wash ) & ( tm=0 ) ) L_Pre-wash )
L_Pre-wash = ( (  & @( washtime=1-pw ) )
               fix X0. ( ( ( & @( rewash ) & ( washtime<4 ) X0 )
                         [] L_Start-washing ) ) )
L_water-in =
  ( ( ( (  & @( check-wl ) & ( hw=0 ) ) L_cold-w
        ( & @( rewash ) & ( washtime<4 ) X0 ) )
    [] ( (  & @( check-wl ) & ( hw=2 ) ) L_hot-w
         ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
    [] ( (  & @( check-wl ) & ( hw=1 ) ) L_warm-w
         ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
L_cold-w = ( (  & @( start-wash ) ) L_washing
             ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_warm-w = ( (  & @( start-wash ) ) L_washing
             ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_hot-w = ( (  & @( start-wash ) ) L_washing
            ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_washing = ( (  & @( washtime=washtime+1 ) ) L_water-out
              ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_water-out = ( (  & @( start-spin ) & ( washtime>1 ) ) L_Spin
                ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_Powder-in = ( ( fill-water ) L_water-in
                ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_Spin = ( & @( Beep-finish ) & ( washtime=4 ) sink
           ( & @( rewash ) & ( washtime<4 ) X0 ) )
```

In the final code, the sink process in L_Idle is replaced by the process L_Check-wait.

Chapter 6

The transformation from abstract to concrete Verilog

In order to move further, in this chapter we introduce the concrete Verilog programs and the the transformation from abstract Verilog to concrete Verilog. The full manual of Verilog can be found at [32, 54] and the behaviour of Verilog programs is described in [2, 7, 11, 23, 30]. A subset of the Verilog syntax is presented in appendix A. In the first section we present a Verilog program to illustrate the Verilog module. The next section describes some transformations from abstract to concrete Verilog. Lastly, we discuss some simple examples.

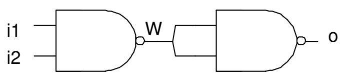
6.1 Verilog program

Verilog is a language based on C where its code is organised as modules. Modules in Verilog are the main units of behaviour. For example, a module behaviour can be specified as:

Behaviourally: $o = \neg(i1 \wedge i2)$

```
module NAND (i1,i2,o);
  input i1, i2;          //inputs
  output o;              //output
  assign o = ~(i1 & i2); //continuous assignment
endmodule
```

Structurally:



```
module AND (i1,i2,o);
  input i1, i2;          //inputs
  output o;              //output
  wire w;                //wire

  NAND NAND1(i1,i2,w);   //module instances
  NAND NAND2(w,w,o);
endmodule
```

In Verilog program, each module has: a name, a port list, declarations, and body of the module. The body consists of one or more items. More examples are shown in appendix B.1.

6.2 The transformation from abstract to concrete Verilog

In this section we only discuss common instructions, other statements like declarations, data structures or operations are not covered. The abstract Verilog is discussed in section 2.2. A subset of the concrete Verilog syntax is presented in appendix A. The transformations are divided into several parts as below:

The primitive commands

- *Skip*: This is a simplest instruction to define as it does nothing.

- *Sink*: Sink is a command to terminate the Verilog program. In a purpose of simulation we can use `$finish` statement to terminate the program.

- *Assignment*: The essential meaning of $v = ex$ is that v takes the value of ex. This is identical to *assignment* statement in Verilog.

Guards

Guards block a process from continuing execution until a certain event occurs, or a number of time units elapse. Guards, as we mentioned in section 2.2, are either time delay (e.g $\#n$) or event of changing values (e.g $\uparrow v, \downarrow v$). Then applying the transformation function ($[\![\,.\,]\!]$) to the guards, we have:

$[\![\#\mathbf{n}]\!] = \#\mathtt{n};$

$[\![\uparrow v]\!] = @(\mathtt{posedge\ v});$

$[\![\downarrow v]\!] = @(\mathtt{negedge\ v});$

$[\![\rightarrow e]\!] = \rightarrow e;$

$@(\mathtt{v})$; is used to wait for a change of v. An other instruction is `wait ex`, it is wait until expression ex is true.

Guard choice

As guarded choice is not defined in Verilog syntax, we shall provide the following algebraic law to eliminate it.

Rule to eliminate the guarded choice (6.2.1)

$$[]_{1\le i\le m}\,(g_i\ P_i)\ \Big[\Big]\ []_{1\le j\le n}\,(h_j\ Q_j)\ =\ []_{1\le i\le m}\,(g_i\ P'_i)\ \Big\|\ []_{1\le j\le n}\,(h_j\ Q'_j)$$

provided that

$$P_i\ =\ P'_i \parallel (\ []_{1\le j\le n}\,(h_j\ Q'_j)\ ),\ 1\le i\le m$$

$Q_j \;=\; (\; []_{1 \le i \le m}\, (g_i\; P_i') \;) \parallel Q_j',\; 1 \le j \le n$

Note that for simplicity, we assume that all guards $g_1, ..., g_m, h_1, ..., h_n$ are disjoint:

Note that from RHS to LHS, it is the expansion law for parallel composition (section 2.2.3). In the next section we will show an example for this rule.

Constructive operators

- *Condition*: The conditional constructor $P \lhd b \rhd Q$ describes a program which behaves like P if the initial value b is true, or like Q if b is false. We have:

  $[\![P \lhd b \rhd Q]\!] =$ `if (b) P else Q`

  If the condition in the form of $P \lhd b \rhd skip$ then the concrete version is `if (b) P`.

  The `if` statement is also used to rewrite the compound guards, $g1$ or $g2$ ... or gn, as follow:

  ```
  if (g1)
  else if (g2)
  ...
  else if (gn)
  ```

- *Iteration*: Iteration constructor of the form: $b * S$, means that the program S is repeatedly executed as long as b is true. The corresponding operator is:

$[\![b \ * \ S]\!] =$ `while (b) S`

We can also use a `forever` loop if b is always true, or a `for` loop if b is a conditional expression based on an integer number (number of cycles is known).

- *Recursion*: A recursion has a form of $\mu X \bullet S$. We discuss here a special case of the recursion, a tail recursion. For example, if a tail recursion has a form of $\mu X \bullet (\ S'; \ X)$ then we can use `always` statement in concrete Verilog to represent it. We have:

  $[\![\mu X \bullet (\ S'; \ X)]\!] =$ `always S'`

6.3 Some examples

Example 1:

Given the following guarded choices:

$$\begin{aligned} P \ = \ & (\ g1\ (\ (@(v=1)\ g3\ skip)\ [\,]\ (g3\ (v=1))\ )) \\ & [\,]\ (g2\ (\ (@(u=1)\ g3\ skip)\ [\,]\ (g3\ (u=1))\ )) \\ & [\,]\ (g3\ (\ (g1\ (v=1))\ [\,]\ (g2\ (u=1))\ )) \end{aligned}$$

We can obtain the following Q by using the elimination rule (6.2.1):

$$Q \ = \ (\ (g1\ (v=1))\ [\,]\ (g2\ (u=1))\ )\ \|\ (g3\ skip)$$

$P \ = \ Q$ can be demonstrated by the following:

Apply the expansion law to Q", we have:

$$\begin{aligned} Q \ = & \Big(g1\ ((v=1)\ \|\ (g3\ skip))\ \Big) \\ & [\,]\ \Big(g2\ ((u=1)\ \|\ (g3\ skip))\Big) \end{aligned}$$

$$[]\ \big(g3\ ((g1\ (v=1))\ []\ (g2\ (u=1))\ )\big)$$

$$=\big(g1\ (\ (@(v=1)\ g3\ skip)\ []\ (g3\ (skip \parallel (v=1)))\ )\big)$$

$$[]\ \big(g2\ (\ (@(u=1)\ g3\ skip)\ []\ (g3\ (skip \parallel (u=1)))\ )\ \big)$$

$$[]\ \big(g3\ (\ (g1\ (v=1))\ []\ (g2\ (u=1))\ )\big)$$

Because of the deduction rule from the parallel expansion section 2.2.3:

$$S \parallel skip \ = \ skip \parallel S \ = \ S$$

then we have:

$$\begin{aligned} Q \ &= \ (\ g1\ (@(v=1)\ g3\ skip\ []\ g3\ (v=1))) \\ &\quad []\ (g2\ (@(u=1)\ g3\ skip\ []\ g3\ (u=1))) \\ &\quad []\ (g3\ (g1\ (v=1)\ []\ g2\ (u=1))) \\ &= \ P \end{aligned}$$

Example 2:

$$L(sc)\ = fix\ X.\ (\ (\ @(v)\ \&\ @(n=n+1)\ )\ \ (\ @(n<10)\ X\ ))$$

Here we have a recursion consisting of two sub-processes. In the first half, a guarded choice v is used to allow the increment of variable n. The second contains no guard, which is automatically performed if the condition $(n < 10)$ is satisfied. A `while` block and an input signal v are used to represent the process as follows:

```
module ex2(v, n);
input v;
output [7:0] n;
reg [7:0] n;

initial begin
```

```
  @( v ) n=n+1;
  while ( n<10 )
    begin
      @( v ) n = n+1;
    end
end
endmodule
```

There are two states and two transitions in this example. The first transition will be taken if value of v is changed. Then the variable n will increased by 1. The second transition will be taken if $n < 10$. This recursion is represented by *while* block where it will waiting for the change of v.

Example 3:

$$L(sc) \;=\; (\; a \;\&\;\; \rightarrow e \;(\; fix\; X.\; (\; e \;(\; \rightarrow f \;\; X \;))))$$

There is a guarded choice a before a recursion. If the guard a is taken, then an event e will be generated. The event e is waiting in the recursion. Other event f will be generated once event e occurs. We can rewrite the corresponding Verilog program as follows:

```
module ex3(a);
input a;
event e, f;

initial begin
  @(a) ->e;
end

always @(e)
  begin
    ->f;
    #1;
  end
endmodule
```

The argument of this module consist of a input variables a and two events e, f. The signal a represent the guarded choice of a transition. When a occurs, the event e will be generated. Then the `always` blocks will be activated and will be executed. After that, the event f will be generated. The `#1;` instruction is used to delay the `always` block, and gives control to other blocks if necessary.

Example 4: In this example we discuss the CD-player program (Verilog process at section 5.1.2). The program of the CD-player consists of two modules to control playing state and track information. In the concrete Verilog version we also write a program for two separate modules. However, these two modules can be called and run in parallel from one main module, or can be composed in other ways. These modules are:

```
module moduleplay(ct, play, pause, stop);
input play, pause, stop;
reg playing;
inout [7:0] ct;
reg   [7:0] ctt;
wire  [7:0] ct = ctt;

initial begin
  playing = 0;
  assign ctt = 1;
end

always @( stop ) begin
  playing = 0;
  ctt = 1;
end

always @( play or pause )
  if ( play )
```

```
      playing = 1;
    else if ( pause )
      playing = 0;

  always begin
    if ( playing ) begin
      //playing
    end
    #1;
  end
  endmodule


  module moduletrack(ct, next, prev, select, maxtrack, trsl);
  input next, prev, select;
  input [7:0] maxtrack, trsl;
  inout [7:0] ct;
  reg   [7:0] ctt;
  wire  [7:0] ct = ctt;

  always @( next ) begin
    if ( ctt < maxtrack )
      ctt = ctt + 1;
  end
  always @( prev) begin
    if ( ctt > 1 )
      ctt = ctt - 1;
  end
  always @( select) begin
    if ( ( ctt > 1 ) && ( ctt < maxtrack ) )
      ctt = trsl;
  end
  endmodule
```

In the first module, the main loop is for `playing` (in the third `always` block). At the beginning all four blocks start to execute, but only the `initial` block terminates. The three `always` blocks are waiting for some events. If signal `play` occurs, it will switch to playing status. If any of `stop` or `pause` signal occurs, then the CD-player will stop playing and change its status. Take note that `#1` is used to give the control to other blocks, as there is an `always` block without an event guard.

In the second module, three `always` blocks are used to control three signals (`next, prev`, and `select`). If any of these signals occurs, the corresponding `always` block will be activated. All of these blocks have event control so we do not need to use time delay, such as `#1`. In these modules we use a `wire` declaration to connect the information of *ct* between modules.

In the examples above show that we can transform abstract Verilog process to concrete Verilog program. However, some difficulties still remain. The most difficult situation is for the recursion. With simple cases like example 2 and 3 (in the previous subsection), we can use `while` loop or `always` block to represent the recursion. Complex programs, which may contain several nested recursions are more difficult because the `always` blocks cannot be written in a hierarchical structure. In this case, we may have to flatten the recursion into several `always` blocks in parallel. Currently, we are formalizing the guarded choice elimination and also replacing the other control structures for abstract Verilog.

Chapter 7

Conclusion

In this thesis we presented the specification of Statechart and abstract Verilog, and then provided a mapping algorithm which is used to translate statecharts into Verilog processes. The main aim of this work is to present the connection between Statecharts and Verilog. We also make use of abstract Verilog to simplify the mapping algorithm, with concrete Verilog being used as actual hardware compilation, where possible.

The main achievement is the construction of a system which maps each input statechart into abstract Verilog. Users use graphical interface to draw their statecharts before our mapping algorithm generates their corresponding Verilog programs. We have discussed also a solution to eliminate the guarded choice and the replacement of the certain structures of abstract Verilog so as to obtain concrete Verilog programs.

There are many approaches in compiling statechart into other languages, including some works that are related to Verilog. However, the powerful features of Statecharts make it difficult to combine into a uniform formalism. Our work follows from the use of formal semantics of statechart and operational semantics of Verilog. This acts as a base to make our mapping algorithm correct and sound.

The mapping of Statecharts into Verilog can be used in hardware design. After translating the input statechart specification into abstract Verilog code, we can proceed to obtain lower level using concrete Verilog, as a prelude to hardware implementation.

Future works

In order to provide the concrete Verilog programs to users, future work include guarded choices elimination and the replacement of the other structures of abstract Verilog, so that the AMSV can generate also concrete Verilog program. This should make our tool especially useful for hardware designer.

Appendix A

The syntax of Verilog

A table bellow in this section is a subset of Verilog syntax.

Table A.1: The syntax of Verilog

module	::=	module <name-of-module> <list-of-ports>; <module-item> endmodule
name-of-module	::=	<identifier>
list-of-ports	::=	| (<port-list>)
port-list	::=	<identifier> | <port-list>, <port-list>
module-item	::=	<parameter-declaration> | <input-declaration> | <output-declaration> | <inout-declaration> | <reg-declaration> | <integer-declaration> | <wire-declaration> | <event-declaration> | <gate-declaration> | <module-instantiation> | <always-statement> | <initial-statement> | <continuous-assign> | <task> | <function> | <module-item> | <module-item>
module-instantiation	::=	<type-of-module> <name-of-instance> <module-agreement>;
parameter-declaration	::=	parameter <range> <list-of-assignments>
range	::=	| [<expression> : <expression>]
list-of-assignments	::=	<param-assignment> | <param-assignment> , <list-of-assignments>
param-assignment	::=	<identifier> = <expression>
input-declaration	::=	input <range> <list-of-variables>;
inout-declaration	::=	inout <range> <list-of-variables>;
output-declaration	::=	output <range> <list-of-variables>;

wire-declaration	::=	wire <range> <list-of-variables>;
		| wire <range> <list-of-assignements>;
integer-declaration	::=	integer <list-of-variables>;
reg-declaration	::=	reg <range> <list-of-variables>;
event-declaration	::=	event <list-of-variables>;
list-of-variables	::=	<name-of-variable>;
		| <name-of-variable>, <list-of-variables>;
name-of-variable	::=	<identifier>
generate-event-statement	::=	-> <event-variable>;
event-variable	::=	<identifier>
initial-statement	::=	initial <statement>
always-statement	::=	always <statement>
event-control-construct	::=	@ <event-variable>
		| @ (<event-expression>)
event-expression	::=	<expression>
		| posedge <expression>
		| negedge <expression>
		| <event-variable> `or` <event-variable>
delay-control-construct	::=	# <number>
		| # <variable>
		| # <expression>
statement-or-null	::=	; | <statement>
statement	::=	if (<expression>) <statement-or-null>
		| if (<expression>) <statement-or-null> else <statement-or-null>
		| case (<expression>) <case-item> endcase
		| casez (<expression>) <case-item> endcase
		| casex (<expression>) <case-item> endcase
		| forever <statement>
		| repeat (<expression>) <statement>
		| while (<expression>) <statement>
		| for (<assignment> ; <expression> ; <assignment>) <statement>
fork-statement	::=	fork <statements> join
statements	::=	<statement>
		| <statements> <statement>
finish-statement	::=	$finish;
stop-statement	::=	$stop;
		| $stop (<expression>);

where, <identifier>: An identifier is any sequence of letters, digits, dollar signs ($), and underscore (_) symbol, except that the first must be a letter or the underscore; the first character may not be a digit or $. Upper and lower case letters are considered to be different. Identifiers may be up to 1024 characters long. Some Verilog-based tools do not recognize identifier characters beyond the 1024th as a significant part of the identifier. Escaped identifiers start with the backslash character (\) and may include any printable ASCII character. An escaped identifier ends with white space. The leading backslash character is not considered to be part of the identifier.

Appendix B

Programs in Verilog

B.1 Simple examples

Example of a counter:

```
module count;
integer count;
initial
begin
  count = 0;
  while (count < 128)
begin
   $display("Count = %d", count);
count = count + 1;
  end
end
endmodule
```

Example of a clock:

```
‘timescale 1ns/1ns
module clock_component (en, clk);
  input en;
  output clk;
  reg clk;
  wire en;
  //
  initial clk = 1'b0;
  always
  begin
    #1000
    if (en == 1) clk = ~clk;
  end
endmodule
```

Example of a office telephone:

```
/* Abstract behavioral system describing a telephone */
module office_phone;
parameter min_conversation=1, max_conversation=30,
          false=0, true=!false;
event ring, incoming_call, answer, make_call, busy;
reg off_hook;
integer seed, missed_calls;
initial begin
  seed=43;    // seed for call duration
  missed_calls=0;
  end

always @ incoming_call      // someone tries to call us
  if (! off_hook) -> ring;  // if not on the phone it rings
  else begin
    -> busy;  // else they get a busy signal
    $display($time," A caller got a busy signal");
    missed_calls = missed_calls + 1;
  end

always @ring begin           // phone is ringing . . .
  $write($time," Ring Ring");// do we want to answer it?
  if ($random & 'b110) begin // yes we will answer it
    -> answer;
    off_hook = true;
    $display(" answered");
  end                 // no we do not want to answer
  else begin          // this phone call
   missed_calls = missed_calls + 1;
   $display(" not answered missed calls =%d",
            missed_calls);
    end
  end

always @make_call
 if (off_hook)
    $display($time," cannot make call phone in use");
  else
  begin
    $display($time," making call");
    off_hook = true;
  end
always wait(off_hook == true) begin // we are on the phone
                                   // wait the call duration
    #($dist_uniform(seed,        // a uniform distribution
       min_conversation,max_conversation))
      off_hook = false;
      $display($time," off phone");
    end
// might wait about 2 hours between making calls
always #($random & 255) -> make_call;

// someone might call in within 4 hours
always #($random & 511) -> incoming_call;

// Simulate two days worth of calls
initial #(60*24*2) $finish;

endmodule
```

B.2 Washing machine example

```
Result:
L_Wash-Ctr || L_Timer-Ctr || L_Water-Ctr || L_Prewash-Ctr ||
  L_Hotwater-Ctr || L_Rinse-Ctr

Where:
L_Wash-Ctr = L_Idle
L_Idle = ( ( Start & @( washing=true ) ) L_Check-wait )
```

```
L_Timer-Ctr =
  fix X1. ( ( ( timer-increase & @( tm=tm+1 ) & ( tm<10 & washing=false ) X1 )
              [] ( timer-decrease & @( tm=tm-1 ) &
                   ( tm>1 & washing=false ) X1 ) ) )
L_Water-Ctr =  fix X2. ( L_Normal )
L_Normal = ( ( Water-pressed & @( wl=1 ) ) L_Half )
L_Half = ( ( Water-pressed & @( wl=2 ) ) ( Water-pressed & @( wl=0 ) X2 ) )
L_Light = ( ( Rinse-pressed & @( rl=1 ) ) L_Medium )
L_Medium = ( ( Rinse-pressed & @( rl=2 ) ) ( Rinse-pressed & @( rl=0 ) X5 ) )
L_Prewash-Ctr =  fix X3. ( L_Pre-w-no )
L_Pre-w-no = ( ( Pre-wash & @( pw=1 ) & ( washing=false ) )
               ( Pre-wash & @( pw=0 ) & ( washing=false ) X3 ) )
L_Hotwater-Ctr =  fix X4. ( L_Cold )
L_Cold = ( ( Hot-water & @( hw=1 ) ) L_Warm )
L_Warm = ( ( Hot-water & @( hw=2 ) ) ( Hot-water & @( hw=0 ) X4 ) )
L_Rinse-Ctr =  fix X5. ( L_Light )
L_Start-washing =
  ( ( ( fill-water & ( washtime!=1 ) ) L_water-in
      ( & @( rewash ) & ( washtime<4 ) X0 ) )
    [] ( (  & @( get-powder-in ) & ( washingtime=1 ) ) L_Powder-in
         ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
L_Wait = ( (  & @( check-pre-wash ) & ( tm=0 ) ) L_Pre-wash )
L_Pre-wash = ( (  & @( washtime=1-pw ) )
               fix X0. ( ( ( & @( rewash ) & ( washtime<4 ) X0 )
                           [] L_Start-washing ) ) )
L_water-in =
  ( ( ( (  & @( check-wl ) & ( hw=0 ) ) L_cold-w
        ( & @( rewash ) & ( washtime<4 ) X0 ) )
      [] ( (  & @( check-wl ) & ( hw=2 ) ) L_hot-w
           ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
      [] ( (  & @( check-wl ) & ( hw=1 ) ) L_warm-w
           ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
L_cold-w = ( (  & @( start-wash ) ) L_washing
             ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_warm-w = ( (  & @( start-wash ) ) L_washing
             ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_hot-w = ( (  & @( start-wash ) ) L_washing
            ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_washing = ( (  & @( washtime=washtime+1 ) ) L_water-out
              ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_water-out = ( (  & @( start-spin ) & ( washtime>1 ) ) L_Spin
                ( & @( rewash ) & ( washtime<4 ) X0 ) )
```

```
L_Powder-in = ( ( fill-water ) L_water-in
                ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_Spin = ( & @( Beep-finish ) & ( washtime=4 ) sink
           ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_Check-wait = ( ( (  & @( timer-cal ) & ( tm>0 ) ) L_Wait )
                   [] ( (  & @( check-pre-wash ) & ( tm=0 ) ) L_Pre-wash ) )
```

Bibliography

[1] J. Ali and J. Tanaka. Converting statecharts into Java code. *In Proceedings of the 5th International Conference on Integrated Design and Process Technology (IDPT99)*, Dallas,Texas, June 1999.

[2] Mark Gordon Arnold. Verilog digital computer design : algorithms into hardware. *Upper Saddle River, NJ : Prentice Hall PTR*, c1999.

[3] I. D. Bates, E. G. Chester, D. J. Kinniment. A statechart based HW/SW codesign system. *International Conference on Hardware Software Codesign archive Proceedings of the seventh international workshop on Hardware/software codesign*, pp. 162 - 166, 1999.

[4] J.-R. Beauvais, T. Gautier, P. Le Guernic, E. Rutten, R. Houdebine. A translation of StateCharts into Signal. *In Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98)*, pp. 52-62, Aizu-Wakamatsu, Japan, March 1998 (IEEE Publ.).

[5] D. Björklund, J. Lilius, and I. Porres. Towards efficient code synthesis from statecharts. *In A. Evans, R. France, and A. M. B. Rumpe, editors, Practical UML-Based Rigorous Development Methods-Countering or Integrating the eXtremists. Workshop of the pUML-Group.*, Lecture Notes in Informatics P-7, Toronto, Canada, October, 2001. GI.

[6] J. P. Bowen, He Jifeng and Xu Qiwen. An Animatable Operational Semantics of the VERILOG Hardware Description Language. *Proc. ICFEM2000: 3rd IEEE International Conference on Formal Engineering Methods, IEEE Computer Society Press*, pp. 199207, York, UK, September 2000.

[7] Michael D. Ciletti. Modeling, synthesis, and rapid prototyping with the Verilog HDL. *Upper Saddler River, N.J. : Prentice Hall*, 1999.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. *MIT Press; 2nd edition* (September 2001).

[9] Alexandre David, M. Oliver Möller and Wang Yi. Formal Verification of UML Statecharts with RealTime Extensions. *In the Proc. of Fundamental Approaches to Software Engineering (FASE 2002)*, LNCS 2306, pp. 218–232, SpringerVerlag, 2002.

[10] Kay Fuhrmann, Jan Hiemer. Formal Verification of Statemate-Statecharts. Report 1998.

[11] Ulrich Golze, Peter Blinzer, Elmar Cochlovius, Michael Schafers, Klaus-Peter Wachsmann. VLSI Chip Design With the Hardware Description Language Verilog: An Introduction Based on a Large RISC Processor Design. *Berlin ; New York : Springer*, 1996.

[12] M. J. C. Gordon. The Semantic Challenge of Verilog HDL. *Proc. Tenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press*, pp. 136145, June 1995.

[13] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comp. Prog.*, vol. 8, pp 231-274, 1987.

[14] D. Harel. On Visual Formalisms. *Communications of the ACM*, Vol. 31, No. 5, pp. 541–530, 1988.

[15] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 4, pp. 293–333, October 1996.

[16] D. Harel, E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7), pp. 31-42, 1997.

[17] J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A Compositional Axiomatization of Statecharts. *Theoretical Computer Science* 101, pp. 289–335, 1992.

[18] Zhu Huibiao, J. P. Bowen and He Jifeng. From Operational Semantics to Denotational Semantics for Verilog. *Proc. CHARME 2001: 11th Advanced*

Research Working Conference on Correct Hardware Design and Verification Methods, Livingston, Scotland, September 2001. Springer-Verlag, LNCS 2144, 2001.

[19] Zhu Huibiao, J. P. Bowen and He Jifeng. Deriving Operational Semantics from Denotational Semantics for Verilog. *Technical Report SBU-CISM-01-16, South Bank University, London, UK*, June 2001.

[20] Daniel C. Hyde. CSCI 320 Computer Architecture Handbook on Verilog HDL. CSCI 320 Handbook on Verilog.

[21] IEEE Standard Hardware Description Language based on the Verilog® Hardware Description Language. *IEEE Standard* 1364-1995, 1995.

[22] Juliano Iyoda and He Jifeng. A Prolog Prototype for the Synthesis of Verilog. *Technical report 237*, UNU, IIST, P.O.Box 3058, Macau, July 2001.

[23] James M. Lee. Verilog Quickstart : a practical guide to simulation and synthesis in Verilog. *Boston : Kluwer Academic Publishers*, 2002.

[24] He Jifeng and Xu Qiwen. An Operational Semantics of a Simulator Algorithm. *Technical Report 204, UNU/IIST*, P.O. Box 3058, Macau, 2000.

[25] He Jifeng and Zhu Huibiao. Formalising Verilog. *Proc. IEEE International Conference on Electronics, Circuits and Systems, IEEE Computer Society Press*, pp. 412415, Lebanon, December 2000.

[26] He Jifeng. An Algebraic Approach to the VERILOG Programming. *in*

the Proc. of 10th Anniversary Colloquium of the United Nations University / International Institute for Software Technology (UNU/IIST), SpringerVerlag, 2002.

[27] J. R. Almeida Júnior, João José Neto. Using Adaptive Models for Systems Description. *Proceedings of the IASTED International Conference Applied Modelling and Simulation*, September, Cairns, Australia, 1999.

[28] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A Compositional Approach to Statecharts Semantics. *NASA/CR2000210086, ICASE Report No.200012*, March 2000.

[29] E. Mikk, Y. Lakhnech, M. Siegel and G. Holzmann. Implementing Statecharts in Promela/SPIN. *in the Proc. of the 2nd IEEE Workshop on IndustrialStrength Formal Specification Techniques, IEEE Computer Society*, 1999.

[30] Zainalabedin Navabi. Verilog digital system design. *New York : McGraw-Hill*, 1999.

[31] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of Statecharts. *in 7th International Conference on Concurrency Theory (CONCUR'96)*, Pisa, Italy, August 1996, LNCS 1119, pp.687–702, SpringerVerlag.

[32] Open Verilog International (OVI). *Verilog Hardware Description Language Reference Manual.*

[33] Gordon J. Pace. Hardware Design Based on Verilog HDL. Ph.D thesis, Oxford University, 1998.

[34] Jan Philipps and Tomohiro Yoneda. Symbolic verification of statecharts. *Technical Report FTS95-37, IEICE.* Technische Universität München, 1995

[35] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. *in the Proc. of the Symposium on Theoretical Aspects of Computer Software*, LNCS 526, pp. 244–264, SpringerVerlag, Berlin.

[36] Qin Shengchao and He Jifeng. An Algebraic Approach to Hardware/Software Partitioning. *Proceedings of ICECS2000*, 273-277, 2000.

[37] Shengchao Qin and WeiNgan Chin. Mapping Statecharts to Verilog for Hardware/Software CoSpecification. *FM03*, Pisa, Italy, Sep. 2003. Lecture Notes in Computer Science, Springer-Verlag.

[38] Hisashi Sasaki. A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine. *DATE99*, pp. 353-357.

[39] G. Schneider and Q.-W. Xu. Towards a formal semantic of Verilog using Duration Calculus. In A. Ravn and H. Rischel, editor, *formal Techniques for Real-Time and Fault Tolerant Systems.* LNCS, Springer-Verlag, 1998.

[40] E. Sekerinski and R. Zurob. From statecharts to code: A tool for the graphical design of reactive systems. *Technical report, McMaster University*, 2001.

[41] E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language, Toronto, Canada, October 2001, volume 2185 of Lecture Notes in Computer Science*, pages 376390. Springer-Verlag, 2001.

[42] E. Sekerinski and R. Zurob. Translating Statecharts to B. in B. Butler, L. Petre, and K. Sere, eds., *Proc. of the 3rd International Conference on Integrated Formal Methods*, Turku, Finland, LNCS 2335, pp. 128144, Springer-Verlag, 2002.

[43] S. Seshia, R. Shyamasundar, A. Bhattacharjee and S. Dhodapkar. A Translation of Statecharts to Esterel. *In J. Wing, J. Woodcock, and J. Davies, eds., FM99: World Congress on Formal Methods, LNCS 1709*, pp. 983–1007, 1999.

[44] A. Sowmya and S. Ramesh. Extending Statecharts with Temporal Logic. *IEEE Transactions on Software Engineering*, Vol. 24, No. 3, March, 1998.

[45] M. von der Beck. A comparison of statechart variants. *In H. Langmaack, W.-P. deRoever, and J. Vytopil, editors, Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science Vol. 863, pp. 128148. Springer Verlag, 1994.

[46] Andrzej Wasowski, Peter Sestoft. On the Formal Semantics of Visual-STATE Statecharts. *IT University Technical Report Series*, TR-2002-19, IT university of Copenhagen, 2002.

[47] Andrzej Wasowski. On Efficient Program Synthesis from Statecharts. *Proc.*

of the Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES03, pp. 163-170, USA, 2003.

[48] Frances Newbery Paulisch. The Design of an Extendible Graph Editor. *Ph.D. Dissertation*, Karlsruhe University, January 1992.

[49] Paul Jay Lucas. A Graphical Editor Proposal for Developing Concurrent, Hierarchical, Finite State Machines. *Technical Report:* UIUCDCS-R-93-1799, Urbana, Illinois.

[50] Stephen Edwards. An Interactive Editor for the Statecharts Graphical Language.
Available at: `http://www1.cs.columbia.edu/~sedwards/sc/overview.html`

[51] DiaGen. The Diagram Editor Generator
Available at: `http://www2.informatik.uni-erlangen.de/DiaGen/`

[52] Honeywell. DOME (the DOmain Modeling Environment)
Available at: `http://www.htc.honeywell.com/dome/`

[53] Jgraphpad. Available at: `http://jgraph.sourceforge.net/`

[54] Chris Satterlee. Verilog Formal Syntax Specification.
Available at: `http://agcad.ict.tuwien.ac.at/info/hdl_gesamt/verilog/`

[55] I-LOGIX Inc. Rhapsody®
Available at: http://www.ilogix.com

[56] Wind River. BetterState®

Available at: `http://www.windriver.com/`

[57] XJ Technologies. AnyStates™

Available at: `http://www.xjtek.com/`

[58] The MathWorks. Stateflow

Available at: `http://www.mathworks.com/products/stateflow/`