Automatic Mapping from Statecharts to Verilog

Viet-Anh Vu Tran[1], Shengchao Qin[2,3] and Wei Ngan Chin[2,3]

[1] Vietsoftware Company, Hanoi, Vietnam
tran.vu.viet.anh@vietsoftware.com
[2] Singapore-MIT Alliance
[3] National University of Singapore
{qinsc,chinwn}@comp.nus.edu.sg

Abstract. Statecharts is a visual formalism suitable for high-level system specification, while Verilog is a hardware description language that can be used for both behavioural and structural specification of (hardware) systems. This paper implements a semantics-preserving mapping from Graphical Statecharts to Verilog programs, which, to the best of our knowledge, is the first algorithm to bridge the gap between Statecharts and Verilog, and can be embedded into the hardware/software co-specification process [19] as a front-end.

1 Introduction

Statecharts [6, 7] is a visual formalism catering for high-level behaviourial specification of embedded systems. Its hierarchical structure, orthogonal and broadcast communication features make the system specification compact and intuitive to understand. It is a very good candidate for executable specification in system design [8]. Moreover, the semantics of Statecharts has been extensively investigated [9, 12, 14, 15, 13] in recent years. Some works also attempt to provide tools for formal verification of Statecharts specifications [4], [14], [20].

Verilog [22], [17] is a widely used language for hardware description in industry [2], [5], [11], [10] 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 [5] 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 work, we will make use of abstract Verilog [10], [18], that is described in the next chapter.

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 [5] in terms of simulation cycles. It has been thoroughly investigated afterwards [25], [24].

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 paper 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 [19]. A mapping algorithm will be given in the following sections, where the soundness has been given in Qin and Chin [18].

The rest of this paper is organized as follows. Sec 2 gives a brief introduction to Statecharts and Verilog. Sec 3 presented the formal definition of the mapping function, followed by its implementation in Sec 4. Sec 5 illustrates our mapping results using two examples, while Sec 6 concludes the paper.

2 Preliminaries

2.1 Formal syntax of statecharts

Statecharts is a 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 [6], [7], [9].

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.

As already mentioned in previous section, Statecharts is extensible by hierarchy, orthogonality or broadcast communication. In this paper, we use the formal syntax of statechart from [7] and [18]. The syntax of Statecharts formula is defined as follows (quoting from [18]):
$\mathcal{S}$: 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.

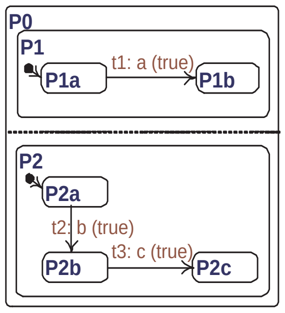
A term-based syntax of statecharts was introduced in [18] and [14], [15]. We re-introduce it here for the benefit of the reader. The set SC is a set of Statecharts terms that is constructed by the following inductively defined functions.

$\texttt{Basic} : \mathcal{S} \rightarrow \texttt{SC}$
$\texttt{Basic}(s) =_{df} |[s]|$
$\texttt{Or} : \mathcal{S} \times [\texttt{SC}] \times \mathcal{T} \rightarrow \texttt{SC}$
$\texttt{Or}(s, [p_1, ..., p_l, ..., p_n], p_l, T) =_{df} |[s : [p_1, ..., p_l, ..., p_n], p_l, T]|$
$\texttt{And} : \mathcal{S} \times 2^{\texttt{SC}} \rightarrow \texttt{SC}$
$\texttt{And}(s, \{p_1, ..., p_n\}) =_{df} |[s : \{p_1, ..., p_n\}]|$

Note that:
– $\texttt{Basic}(s)$: denotes a basic statechart named s.
– $\texttt{Or}(s, [p_1, ..., p_l, ..., p_n], p_l, \mathcal{T})$: represents an 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.
– $\texttt{And}(s, \{p_1, ..., p_n\})$ is an And-statechart named s, which contains a set of orthogonal (concurrent) sub-states $\{p_1, ..., p_n\}$.

In this paper we use sub-state interchangeable as children of Or-state. Correspondingly, we use children and region of And-state interchangeably. For statecharts that we adopted in this work, we shall assume that each And-state will have at least two regions. Furthermore, each region shall be an Or-state.

We shall take the textual representation of statecharts as input data for our core mapping program. Our front-end algorithm will translate graphic charts to textual representation automatically. As an example, we give below a simple graphical Statechart and its corresponding textual representation.



```
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 ]|

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

Fig. 1. A simple example of a Statechart and its textual representation.

2.2 Verilog

Verilog is a hardware description language that has been widely used in industry. Although the Verilog IEEE standard [22] was released around ten years ago, the formal semantics based on simulation cycles [5] has not been well-investigated until recently, e.g. [11], [10]. In our work, we shall use a behaviourial subset of Verilog introduced in [10] and [18]. 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 non-trivial. 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
   $P \ ::= \ S \ \mid \ P \parallel \ P$
   where, S is a sequential process.
2. Sequential process can be formally described as following
   $S \ ::= PC$ (primitive command) $\mid \ S; S$ (sequential composition)
   $\mid \ s \ \lhd \ b \ \rhd \ S$ (condition) $\mid \ b \ * \ S$ (iteration)
   $\mid \ (b \& g \ S)$ [] ... [] $(b \& g \ S)$ (guarded choice) $\mid \ fix \ X \bullet S$ (recursion)

   where, b is boolean condition, and
   $PC ::= skip \mid sink \mid \perp \mid \ \rightarrow \ \eta$ (output event) $\mid v \ = \ ex$ (assignment)
   $g \quad ::= \rightarrow \ \eta \mid @(x = v)$ (assignment guard))
   $\mid \#1$ (time delay) $\mid eg$ (event control)
   $eg \quad ::= \eta \mid eg \ \& \ eg \mid eg \ \& \ \neg eg$
   $\eta \quad ::= \uparrow \ v$ (value rising) $\mid \ \downarrow \ v$ (value falling) $\mid \underline{e}$ (a set of abstract events)

Recall that a Verilog program can only be a parallel process at the top level, a sequential process cannot contain a parallel process. However, most real systems contain many parallel processes possibly organised hierarchically. To solve this restriction, we shall use algebraic laws [10] to expand a parallel process into a sequential one.

Here are some simple code examples:

– $(e \ \& \ (\rightarrow f) \ sink)$ [] $(g \ \& \ (\rightarrow h) \ sink)$
– $\mu X \bullet (e \ (f \ X) \ )$
– $(a \ \& \ (\rightarrow e) \ sink) \parallel (b \ \& \ (\rightarrow f) \ sink)$

3 Semantic-Preserving Mapping

Our algorithm that takes as input graphical statecharts and generates as output Verilog code is based on the theoretical result presented in [18]. 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 [18] which produces result based on the type of the source statechart:

Definition of mapping function L:

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

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.,

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

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

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

– 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*.

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

where

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

and

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

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* : $\texttt{SC} \to N$ to calculate the "or–depth" of a statechart, which is defined as follows:

- for a statechart $sc = |[s]|$ constructed by `Basic`, *or-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 $m < k \leq n$.

Note that $T^*(p)$ 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 as 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:

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

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 $sc_{[l \mapsto p_m]} =_{df} |[s : [p_1, ..., p_n], p_m, T]|$

4 Implementation

Our implementation consists of two parts: a statechart editor (called Statechart_E, is a stencil of MS Visio) and a mapping program from statechart into abstract Verilog (called AMSV-Automatic Mapping of Statechart into Verilog).

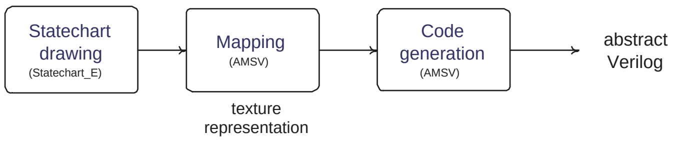


Fig. 2. Structure of the implementation.

Fig. 2 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.

4.1 Statechart editor

Statechart_E 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.

Features of Statechart_E:

– A menu named *Statechart* is added to the menu bar of Visio. 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.

– 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.
– 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. Note that the *generate* function will read all components in all pages of the statechart.

4.2 AMSV - Core mapping program

The second part, called AMSV (Automatic Mapping of Statechart into Verilog), is essentially a Java program.

DFS algorithm As presented in section 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 [3] 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. 3 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.

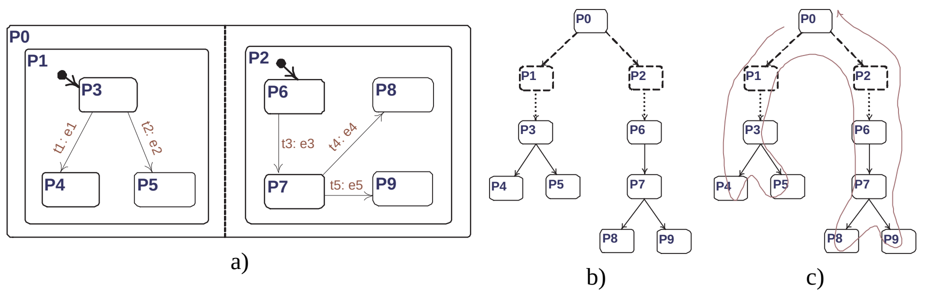


Fig. 3. Hierarchy tree. a) Statechart example, b) hierarchy tree, and c) DFS route.

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. 3, 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. 3 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.

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.

Parallel expansion Recall from early discussion in Sec 2, we shall 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 [23].

5 Examples

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

5.1 CD-player

Specification Fig. 4 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.

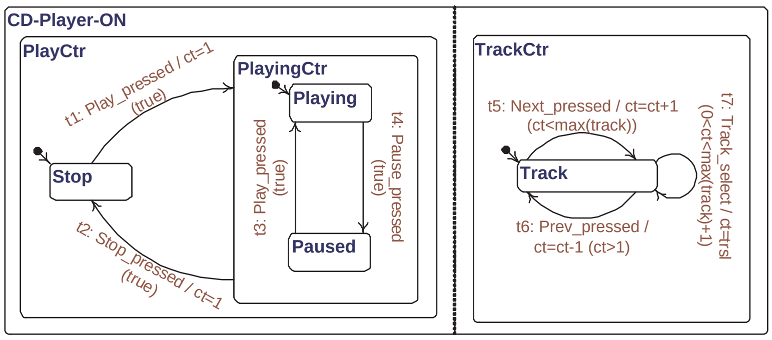


Fig. 4. CD player with track information (ct).

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 >
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.

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

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 shows the user interface of the washing machine. Fig. 6 gives the statechart specification of the washing machine corresponding to the interface, while Fig. 7 zooms into the sub-state `Washing-Ctr`. Statechart in Fig. 6 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 with black circles ($hw = 0$, $rl = 0$, $wl = 0$, and $pw = 0$) and default timer is 0.

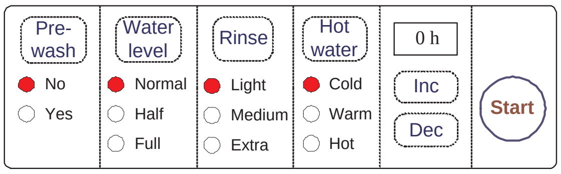


Fig. 5. Interface of the washing machine.

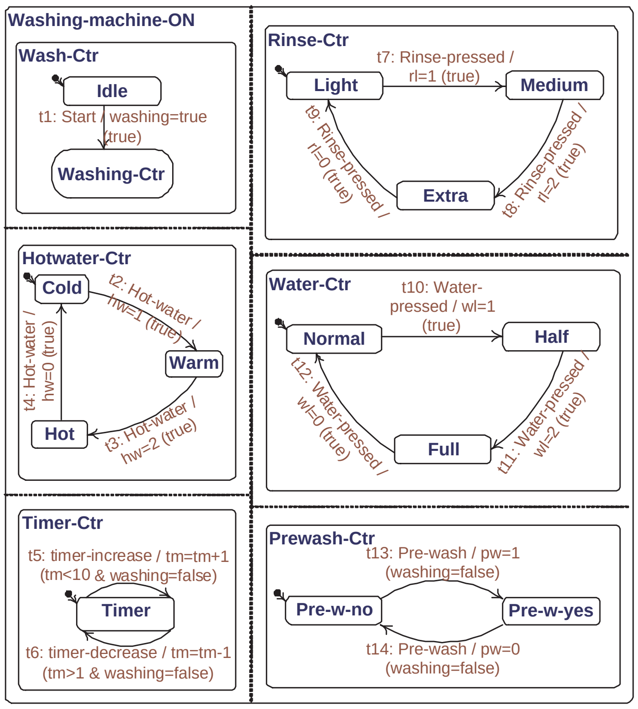

Washing-machine-ON
Wash-Ctr
Idle
t1: Start / washing=true (true)
Washing-Ctr
Rinse-Ctr
t7: Rinse-pressed / rl=1 (true)
Light
Medium
Extra
t9: Rinse-pressed / rl=0 (true)
t8: Rinse-pressed / rl=2 (true)
Hotwater-Ctr
Cold
t2: Hot-water / hw=1 (true)
Warm
t3: Hot-water / hw=2 (true)
Hot
t4: Hot-water / hw=0 (true)
Water-Ctr
t10: Water-pressed / wl=1 (true)
Normal
Half
Full
t12: Water-pressed / wl=0 (true)
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)

Fig. 6. Main statechart of a washing machine.

The `Washing-Ctr` is an `Or`-state as given in Fig. 7. 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 *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.
– $washtime > 3$: finish.

Upon finishing, the machine beeps to inform the user.
The textual representation generated from Statechart_E is printed in [23].

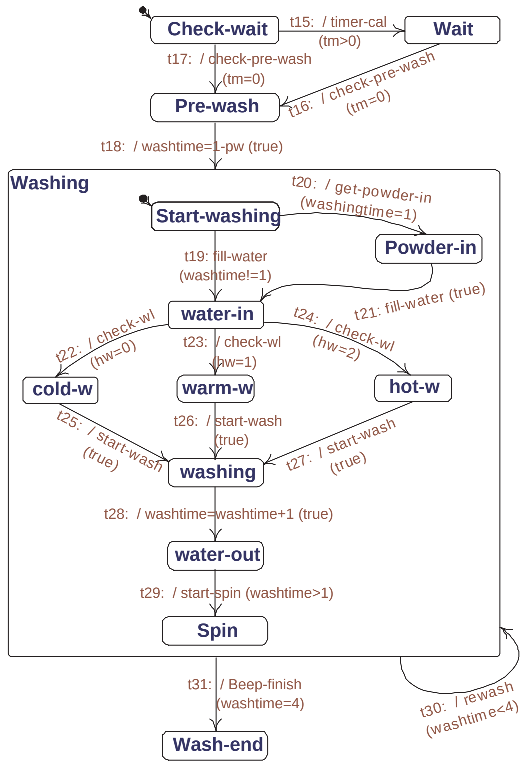


Fig. 7. Statechart of `Washing-Ctr` in the washing machine.

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.
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`.

6 Conclusion

In this paper we proposed an automatic mapping algorithm to translate high-level Statecharts into low-level Verilog specifications. Our algorithm has been proved sound earlier [18].

The system that we have built in Java provides a graphical interface for users to draw their statecharts in MS Visio. Our mapping algorithm thus translates the graphical representation into a textual representation, and then generates the corresponding Verilog programs.

Some of related works on connecting Statecharts with other formalisms are presented in [1, 4, 16, 21, 20]. Beauvais et.al. [1] and Seshia et.al. [21] translate STATEMATE Statecharts to synchronous languages *Signal* and *Esterel* respectively, aiming to use supporting tools provided in the target formalisms for formal verification purposes. However, all these translations are based on the informal semantics [9] lacking correctness proofs. The authors of [4, 16] transform variants of Statecharts into hierarchical timed automata and use tools (UPPAAL, SPIN) to model check Statecharts properties. More recently, a translation from Statecharts to B/AMN is reported in [20]. However, no correctness issue has been addressed. In comparison, the translation from Statecharts to Verilog in this paper aims at code generation for system design. The mapping function that we implement in this paper is constructed based on formal semantics for both the source and target formalisms and has been proven to be semantics-preserving [18].

Our compilation from Statecharts into Verilog can be used as a front-end of hardware design or hardware/software co-design. After translating the input statechart specification into abstract Verilog code, we can proceed to obtain lower level descriptions, as a prelude to hardware implementation, or we can pass the Verilog specification to a hardware/software partitioning system [19].

In order to provide the concrete Verilog programs to users, future works 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.

References

1. J.-R. Beauvais, et. al. A Translation of Statecharts to Signal/DC+. Technical report, IRISA, 1997.
2. J. P. Bowen, J.-F. He, and Q.-W. Xu. An Animatable Operational Semantics of the VERILOG Hardware Description Language. In *Proc. ICFEM2000: 3rd IEEE International Conference on Formal Engineering Methods, IEEE Computer Society Press*, York, UK, September 2000.
3. T. H. Cormena, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press; 2nd edition, September 2001.
4. A. David, M. Oliver Möller, and Wang Y. Formal Verification of UML Statecharts with Real-time Extensions. In *Proc. of Fundamental Approaches to Software Engineering*, number 2306 in Springer LNCS, 2002.
5. M. J. C. Gordon. The Semantic Challenge of Verilog HDL. In *Proc. Tenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press*, pages 136–145, June 1995.

6. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.
7. D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5), 1988.
8. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7), 1997.
9. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), October 1996.
10. J.-F. He. An Algebraic Approach to the VERILOG Programming. In *Proc. of 10th Anniversary Colloquium of the United Nations University / International Institute for Software Technology (UNU/IIST)*. Springer, 2002.
11. J.-F. He and H. Zhu. Formalising Verilog. In *Proc. IEEE International Conference on Electronics, Circuits and Systems, IEEE Computer Society Press*, Lebanon, December 2000.
12. J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A Compositional Axiomatization of Statecharts. *Theoretical Computer Science*, 101, 1992.
13. Q. Long, Z.Y. Qiu, and S.C. Qin. The Equivalence of Statecharts. In *International Conference on Formal Engineering Methods*, number 2885 in Springer LNCS, Singapore, November 2003.
14. G. Lüttgen, M. von der Beeck, and R. Cleaveland. A Compositional Approach to Statecharts Semantics. Technical Report 200012, NASA/CR2000210086, ICASE Report, March 2000.
15. A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of Statecharts. In *7th International Conference on Concurrency Theory (CONCUR'96)*, number 1119 in Springer LNCS, Pisa, Italy, August 1996.
16. E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing Statecharts in Promela/SPIN. In *the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*. IEEE Computer Society, 1999.
17. Open Verilog International (OVI). *Verilog Hardware Description Language Reference Manual*.
18. S.C. Qin and W.N. Chin. Mapping Statecharts to Verilog for Hardware/Software Co-Specification. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods: International Symposium of Formal Methods Europe*, volume 2805, pages 282–299. Springer, 2003.
19. S.C. Qin, J.F. He, Z.Y. Qiu, and N.X. Zhang. Hardware/Software Partitioning in Verilog. In *International Conference on Formal Engineering Methods*, number 2495 in Springer LNCS, Shanghai, China, October 2002.
20. E. Sekerinski and R. Zurob. Translating Statecharts to B. In B. Butler, L. Petre, , and K. Sere, editors, *Proc. of the 3rd International Conference on Integrated Formal Methods*, number 2335 in Springer LNCS, Turku, Finland, 2002.
21. S. Seshia, R. Shyamasundar, A. Bhattacharjee, and S. Dhodapkar. A Translation of Statecharts to Esterel. In J. Wing, J. Woodcock, and J. Davies, editors, *FM99: World Congress on Formal Methods*, number 1709 in Springer LNCS, 1999.
22. IEEE Standard. *IEEE Standard Hardware Description Language based on the Verilog® Hardware Description Language*. 1995.
23. V.-A. V. Tran. Automatic Mapping from Statecharts to Verilog. Master's Thesis, School of Computing, The National University of Singapore, 2004.
24. H. Zhu, J. P. Bowen, and J.-F. He. Deriving Operational Semantics from Denotational Semantics for Verilog. Technical report, Technical Report SBU-CISM-01-16, South Bank University, London, UK, June 2001.
25. H. Zhu, J. P. Bowen, and J.-F. He. From Operational Semantics to Denotational Semantics for Verilog. In *Proc. CHARME 2001: 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, number 2144 in Springer LNCS, Livingston, Scotland, September 2001.