Mapping Statecharts to Verilog for Hardware/Software Co-Specification

Shengchao Qin[1] and Wei-Ngan Chin[1,2]

[1] Singapore-MIT Alliance, National University of Singapore
[2] School of Computing, National University of Singapore
{qinsc,chinwn}@comp.nus.edu.sg

Abstract. Hardware-Software co-specification is a critical phase in co-design. Our co-specification process starts with a high level graphical description in Statecharts and ends with an equivalent parallel composition of hardware and software descriptions in Verilog. In this paper, we investigate the Statecharts formalism by providing it a formal syntax and a compositional operational semantics. After that, we design a semantics-preserving mapping function to transform a Statecharts description into Verilog specification. We can combine this mapping with our previous formal partitioning process so as to form a more complete and automated co-specification process.

Keywords: Statecharts, Verilog, operational semantics, homomorphism

1 Introduction

The design of a complex control system is ideally decomposed into a progression of related phases. It starts with an investigation of properties and behaviours of the process evolving within its environment, and an analysis of the requirement for its safety performance. From these is derived a specification of the electronic or program-centred components of the system. The process then may go through a series of design phases, ending in a program expressed in a high level language. After translation into a machine code of a chosen computer, it can be executed at a high speed by electronic circuity. In order to achieve time performance required by the customer, additional application-specific hardware devices may be needed to embed the computer into the system which it controls.

Classical circuit design methods resemble the low level machine language programming methods. These methods may be adequate for small circuit design, but not adequate for circuits that perform complicated algorithms. Industry interests in the formal verification of embedded systems are gaining ground since an error in a widely used hardware device can have adverse effect on profits of the enterprise concerned. A method with great potential is to develop a useful collection of proven equations and other theorems, to calculate, manipulate and transform a specification formulae to the product.

Hardware/software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of the co-design process is the hardware/software co-specification, which starts from a high level system specification and ends with a pair of sub-specifications representing resp. hardware and software. Our previous work ([17]) proposes a formal partitioning algorithm which splits a Verilog source program into hardware and software specifications. The partitioning correctness is verified using algebraic laws developed for the Verilog hardware description language. This algebraic approach has also been demonstrated in our earlier work [15, 16]. One of advantages of this approach is that it ensures the correctness of the partitioning process. Moreover, it optimises the underlying target architecture, and facilitates the reuse of hardware devices.

In this paper, we bridge the gap between the high level specification in Statecharts and the Verilog source program by defining a mapping function between the two formalisms. Through this work, the overall co-specification process can be automated, as illustrated in Fig.1. Two key contributions of the present paper are:

– we propose a formal operational semantics for a subset of Statecharts with data states, which adopts an asynchronous model and supports true concurrency;
– we define a formal mapping function which transforms a Statechart specification into a Verilog program. We show that the target program after mapping preserves the semantics of the source specification.

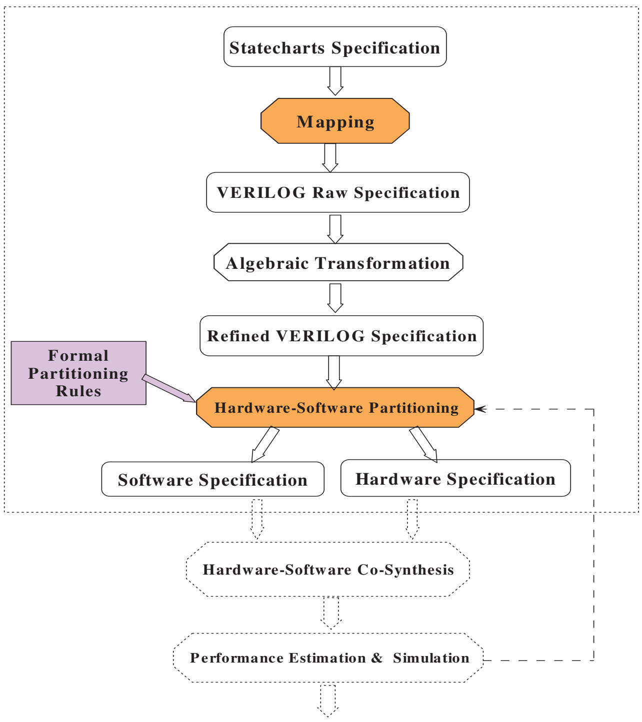

Fig. 1. HW-SW Co-Specification

The mapping process can be integrated with our previous formal partitioning algorithm so as to form an automated hardware-software co-specification process for hardware-software co-design, as summarised in Fig.1. The remainder of this paper is organised as follows. Section 2 first gives a formal (text-based) syntax for Statecharts with data states and proposes a compositional operational semantics for it afterwards. Section 3 introduces a subset of Verilog for behaviourial specification. We build a mapping function from Statecharts into Verilog and prove that it is a homomorphism between the two formalisms in Section 4. Related works together with a simple discussion and conclusion follow afterwards.

2 Operational Semantics for Statecharts

The graphical language of Statecharts as proposed by David Harel ([4]) is suitable for the specification and modeling of reactive systems. While the (graphical) syntax of the language has been formulated quite early, the definition of its formal semantics proved to be more difficult than originally expected. As discussed in [14], these difficulties may be explained as resulting from several requirements that seem to be desirable in a specification language for reactive systems, but yet conflict with one another in some interpretations. This may be why there exist more than twenty variants of Statecharts ([21]), each of which can be regarded as a subset of the originally expected language. The version discussed in [6] for STATEMATE is rather large and powerful; however, their operational semantics is neither formal nor compositional. The work presented in [11] provides a compositional semantics for Statecharts, but does not contain data states. Hooman *et.al* ([9]) proposes a denotational semantics based on histories of computation. Following this line, [20] attempts to link the denotational semantics of Statecharts with temporal logic, so as to support formal verification. All these works adopt a synchronous model of time, which is simpler to understand and formalise, but less powerful than the asynchronous model.

Our version of Statecharts involves data items. The model we adopt is the asynchronous model, which is more powerful for specifying and modeling complex systems. Our formal operational semantics comprises the following features.

- It is *compositional*, which implies that inter-level transitions and state references have been dropped. The history mechanism has also been ignored.
- It adopts an asynchronous time model, in which a macro-step (comprising a sequence of micro-steps) occurs instantaneously. This model supports *perfect synchrony hypothesis* and also supports state refinement in top-down design.
- It reflects the *causality* of events.
- To be more intuitive, our semantics obeys *local consistency*, rather than *global consistency*. That is, the absence of an event may lead to itself directly or indirectly in the same macro-step.
- Instantaneous states are allowed, but each state cannot be entered twice or more at the same instant of time. [1]

[1] For simplicity, this checking is omitted in our semantics. We can include it by keeping records of the states that are passed so far in the current macro-step and prevent a former state from being re-entered in each macro-step.

– It covers the data-state issues of Statecharts, allowing assignments in state transitions.
– It supports true concurrency.

In this paper, timeout events are not included and this aspect is left as future work.

In what follows we give a formal syntax for Statecharts, and afterwards investigate its operational semantics thoroughly.

2.1 A Formal Syntax of Statecharts

Quoting from [5], *state charts = finite-state diagrams + depth + orthogonality + broadcast communication*. This equation indicates the typical features of the Statecharts formalism:

– It is an extension of conventional finite state machines (Mealy machine).
– It provides natural notion of depth. A state can either be a basic one, or of a hierarchical structure, inside which some other states are treated as its substates.
– It supports the modeling of concurrency. A state may contain several states as its concurrent components. This feature also helps to avoid state explosion.
– It provides the broadcast communication mechanism. Unlike CSP or CCS, its output events are asynchronous, and can be broadcast to any receiver without waiting. However, its input events are synchronous, and are blocked until the arrival of the corresponding output events. Such a communication mechanism is similar to Verilog.

In order to formalise the syntax of Statecharts, we introduce the following notations.

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

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

$\sigma : Var \to Val$ 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 $\bar{v}$ is $\sigma(\bar{v})$.

$\mathcal{T}$: the 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.

Similar to [12, 11], we give a term-based syntax for Statecharts. The set SC of Statecharts terms is constructed by the following inductively defined functions.

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

Some informal explanations follow:

- $\mathsf{Basic}(s)$ denotes a basic statechart named s.
- $\mathsf{Or}(s, [p_1, \cdots, p_l, \cdots, p_n], p_l, T)$ represents an Or-statechart with a set of substates $\{p_1, \cdots, p_n\}$, where p_1 is the default substate, p_l is the active substate, T is composed of all possible transitions among immediate substates of s.
- $\mathsf{And}(s, \{p_1, \cdots, p_n\})$ is an And-statechart named s, which contains a set of orthogonal (concurrent) substates $\{p_1, \cdots, p_n\}$.

2.2 Operational Transition Rules

The configuration of computation is defined by a triple $\langle p, \sigma, E_{in} \rangle$, where

- p is the syntax of the statechart of interest.
- σ gives the snapshot of data items.
- E_{in} denotes the current environment of active events.

The behaviour of a statechart is composed of a sequence of macro-steps, each of which comprises a sequence of micro-steps. A statechart may react to any stimulus from the environment at the beginning of each macro-step by performing some enabled transitions and generating some events. This may fire other state transitions and lead to a chain of micro-steps without advancing time. During this chain of micro-steps, the statechart does not respond to any potential external stimulus. When no more internal transitions are enabled, the clock tick transition will occur by emptying the set of active events and advancing time by one unit.

We explore a set of transition rules comprising state transitions and time advance transitions.

At any circumstance, what a basic statechart can do is to advance time by a clock tick.

1. $\langle |[s]|, \sigma, E \rangle \xrightarrow{\surd} \langle |[s]|, \sigma, \emptyset \rangle$

If a transition between two immediate substates of an Or-statechart is enabled and the transition condition is true in current circumstance, it can be performed.

$$\textbf{2.}\ \frac{\begin{array}{c} p = |[s : [p_1, \cdots, p_n], p_l, T]| \\ \tau \in \mathit{En}(p, E) \ \wedge \ \sigma(b) \end{array}}{\langle p, \sigma, E \rangle \xrightarrow{\tau\&b} \langle p_{[l \mapsto \mathbf{a2d}(\mathit{tgt}(\tau))]}, \sigma', (E - \mathit{trig}^+(\tau)) \cup a^e(\tau) \rangle}$$

where

$\mathit{src}(\tau)$ and $\mathit{tgt}(\tau)$ denote, respectively, the source and target state of transition τ.

$a^e(\tau) \subseteq \Pi$ represents all events generated by transition τ, whereas $a^a(\tau)$ denotes a single assignment action $v = ex$ generated by τ. No loss of general results since a sequence of instantaneous assignment statements can be transformed into a single one. This changes the data state from σ to $\sigma' = \sigma \oplus \{v \mapsto \sigma(ex)\}$.

$\mathit{En}(p, E)$ comprises all transitions among substates of p being enabled by events in E. It can be generated by the following definition.

$$\begin{array}{l} \tau \in \mathit{En}(|[s : [p_1, \cdots, p_n], p_l, T]|, E) \quad \textbf{iff} \\ \tau \in T \ \wedge \ \mathit{src}(\tau) = p_l \ \wedge \ \mathit{trig}^+(\tau) \subseteq E \ \wedge \ \overline{\mathit{trig}^-(\tau)} \cap E = \emptyset. \end{array}$$

where $trig^{+}(\tau)$ and $trig^{-}(\tau)$ represent respectively the positive events and the negated events from τ.

The function $\mathbf{a2d}(p)$ changes the active substate of p into its default substate, and the same change is applied to its new active substate.

$$\begin{aligned}
&\mathbf{a2d}(|[s]|) =_{df} |[s]| \\
&\mathbf{a2d}(|[s:[p_1,\cdots,p_n],p_l,T]|) =_{df} |[s:[p_1,\cdots,p_n],\mathbf{a2d}(p_1),T]| \\
&\mathbf{a2d}(|[s:\{p_1,\cdots,p_n\}]|) =_{df} |[s:\{\mathbf{a2d}(p_1),\cdots,\mathbf{a2d}(p_n)\}]|
\end{aligned}$$

The substitution $p_{[l\mapsto p_m]}$ for an Or-statechart $p=|[s:[p_1,\cdots,p_n],p_l,T]|$ is defined by

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

Discussion: in rule **2**, those events that are used to trigger τ are consumed by τ and will no longer exist. This mechanism looks intuitive and reasonable and can help to prevent incorrect looping. Consider an example given in Fig. 2 (a). When the first event e from the environment comes, the transition τ_1 is performed and the active substate is migrated from p_1 to p_2. This will not move back to p_1 until next event e occurs, as under normal expectation. Earlier work ([14]) suggests a different treatment, where active events are kept active during all micro-steps in a macro-step, where they may be reused many times. $\square$

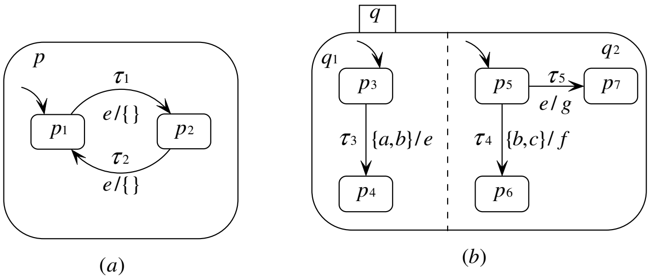

Fig. 2. Example Statecharts (a) and (b)

The transitions in Statecharts are considered hierarchically. If no transitions among immediate substates of an Or-statechart are enabled, an enabled (inner) transition for the active substate may be performed instead. This consideration is carried out inductively as highlighted in rule **3**.

$$\textbf{3.}\ \frac{\begin{array}{c} p=|[s:[p_1,\cdots,p_n],p_l,T]| \\ En(p,E)=\emptyset \\ \langle p_l,\sigma,E\rangle \xrightarrow{\tau\&b} \langle p_l',\sigma',E'\rangle \end{array}}{\langle p,\sigma,E\rangle \xrightarrow{\tau\&b} \left\langle p_{[l\mapsto p_l']},\sigma',(E-trig^{+}(\tau))\cup a^{e}(\tau)\right\rangle}$$

If no transition is enabled for an OR-statechart, time advances, as shown below.

$$\textbf{4.}\quad \frac{En^*(p, E) = \emptyset}{\langle p = \|[s : [p_1, \cdots, p_n], p_l, T]\|, \sigma, E\rangle \xrightarrow{\surd} \langle p, \sigma, \emptyset\rangle}$$

The premise indicates that no transitions in p can be triggered by E. The set of transitions that are enabled at multiple levels is defined as follows.

$En^*(\|[s]\|, E) =_{df} \emptyset$, *for any basic state* $\|[s]\|$;
$En^*(p = \|[s : [p_1, \cdots, p_n], p_l, T]\|, E) =_{df} En(p, E) \cup En^*(p_l, E)$;
$En^*(p = \|[s : \{p_1, \cdots, p_n\}]\|, E) =_{df} \bigcup_{1 \le i \le n} En^*(p_i, E)$.

For a parallel statechart, variables are shared by all orthogonal components. However, each variable can only be modified by one component. We use $WVar(p)$ to denote the set of variables that can be modified by a statechart p.

It is natural and intuitive to accept that several transitions allocated in orthogonal components may be fired simultaneously. This implies that they can be performed in a truly concurrent way. However, we have to write the transition rule for parallel statecharts carefully. Let us look at the statechart in Fig. 2 (b). Suppose the external stimulus is $E = \{a, b, c\}$, which will fire both τ_3 and τ_4 at the same moment. Under rule **2**, performing either of them will prevent another from happening since the common event b is consumed by the performed transition. This contradicts the above intuitive explanation.

We propose a more reasonable way in which simultaneously enabled transitions are allowed to occur concurrently within And-charts. In the following rule, we suppose $i_1, \cdots, i_n$ is a permutation of $1, \cdots, n$.

$$\textbf{5.}\quad \frac{\begin{array}{c} p = \|[s : \{p_1, \cdots, p_n\}]\|, \textit{ all } p_i \textit{ are constructed by } \mathsf{Basic} \textit{ or } \mathsf{Or} \\ \langle p_{i_k}, \sigma, E\rangle \xrightarrow{\tau_{i_k} \& b_{i_k}} \langle p'_{i_k}, \sigma_{i_k}, E_{i_k}\rangle, \textit{ for all } 1 \le k \le m \\ En^*(p_{i_k}, E) = \emptyset, \textit{ for all } m < k \le n \\ WVar(p_i) \cap WVar(p_j) = \emptyset, \textit{ for all } i, j, \textit{ where } i \ne j \\ \sigma' = \sigma_{i_1} \oplus \cdots \oplus \sigma_{i_m} \\ E' =_{df} (E - \bigcup_{1 \le i \le m} trig^+(\tau_{i_k})) \cup \bigcup_{1 \le i \le m} a^e(\tau_{i_k}) \end{array}}{\langle p, \sigma, E\rangle \xrightarrow{\&_{1 \le k \le m}(\tau_{i_k} \& b_{i_k})} \langle \|[s : \{p'_{i_1}, \cdots, p'_{i_m}, p_{i_{m+1}}, \cdots, p_{i_n}\}]\|, \sigma', E'\rangle}$$

In this rule, the overall transition that the And-chart p performs involves several simultaneously enabled transitions $\tau_{i_k}(1 \le k \le m)$ which are performed respectively by components $p_{i_k}(1 \le k \le m)$. Other components p_{i_k} $(m < k \le n)$ are not involved in this transition.

A time advance transition will take place if all orthogonal components agree to do so.

$$\textbf{6.}\quad \frac{En^*(p_i, E) = \emptyset,\ i = 1, \cdots, n}{\langle p = \|[s : \{p_1, \cdots, p_n\}]\|, \sigma, E\rangle \xrightarrow{\surd} \langle p, \sigma, \emptyset\rangle}$$

3 Verilog and Its Operational Semantics

Hardware description languages (HDLs) are widely used to express designs at various levels of abstraction in modern hardware design. A HDL typically contains a high level subset for behaviour description, with the usual programming constructs such as assignments, conditionals, guarded choices and iterations. It also has appropriate extensions for real-time, concurrency and data structures for modeling hardware. VHDL and Verilog ([10]) are two contemporary HDLs that have been widely used for years. Although the formal semantics of VHDL has been studied quite thoroughly, that of Verilog has been ignored until recently ([3, 7, 8, 22, 23]). However, it is reported that Verilog has been more widely used in industry (especially in US)([3]).

What we shall use is a simple version of Verilog with some notational extension (as discussed in [7]) which contains the following categories of syntactic elements.

1. A Verilog program can be a sequential process or a parallel program made up of a set of sequential processes.

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

2. A sequential process in Verilog can be any of the following forms.

$$\begin{aligned} S ::= \ & PC(\text{primitive command}) \mid S; S\,(\text{sequential composition}) \\ & \mid S \lhd b \rhd S\,(\text{conditional}) \mid b * S\,(\text{iteration}) \\ & \mid (b\&g\,S)[\!]\ldots[\!](b\&g\,S)\,(\text{guarded choice}) \mid \mu X \bullet S\,(\text{recursion}) \end{aligned}$$

where b is a boolean condition, and

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

Although Verilog has been standardised ([10]) and widely used in industry, its precise semantics is still lacking. Some recent work ([7, 8, 22, 23]) attempted to address its formal semantics issues from different points of views. The most recent work ([7]) discussed these distinct views, especially the algebraic and operational semantics for Verilog, and explored the underlying links between them.

The subset of Verilog we adopt is quite similar to that proposed by He ([7]). However, there are some different treatments between our version and He's version. We include explicitly the possible context environment of active events in our configuration, and change the operational rules for the parallel constructs. This facilitates our semantic mapping from Statecharts into Verilog, and does not change the observable behaviour of a program.

In our operational semantics of Verilog, transitions are of the form $S \xrightarrow{l} S'$. The configuration S describes the state of an executing mechanism of Verilog programs together with the environment of active events before an action l, whereas S' describes that immediately after. They are identified as triples $\langle P, \sigma, E \rangle$, where

- P is a program text, representing the rest of the program that remains to be executed.
- σ : *Var* $\to$ *Val* records the data state.
- E is the current set of active events.

A label l denotes a transition from state S to S'. It can be a clock tick event $\surd$, or a compositional event possibly with three conjunctive parts: $b\&g^i\&g^o$ representing the enabling condition, the set of events consumed, and the set of events generated, respectively.

Now we present a critical subset of transition rules which are relevant to our transformation from Statecharts into Verilog.

The primitive *sink* can do nothing but advance time by a clock tick.

$$\langle \mathit{sink}, \sigma, E\rangle \xrightarrow{\surd} \langle \mathit{sink}, \sigma, \emptyset\rangle$$

The guarded choice construct

$$P = (b_1\&g_1^i\&g_1^o\, P_1) [\!] \ldots [\!] (b_n\&g_n^i\&g_n^o\, P_n)$$

can take a guarded transition if that guard is enabled.

$$\frac{\sigma(b_k) \wedge (E \vdash g_k^i),\ \mathit{for\ some}\ k}{\langle P, \sigma, E\rangle \xrightarrow{b_k\&g_k^i\&g_k^o} \langle P_k, \sigma', E - e^c(g_k^i) \cup e^g(g_k^o)\rangle}$$

where $E \vdash g^i$ indicates that the input guard g^i is enabled by E. This is defined as:

$$E \vdash \underline{e}_1\& \cdots \&\underline{e}_m\&\neg\underline{e}'_1\& \cdots \&\neg\underline{e}'_n =_{df} \wedge_{1\le i\le m}(\underline{e}_i \subseteq E) \wedge \wedge_{1\le i\le n}(\underline{e}'_i \cap E = \emptyset)$$

Also, $e^c(g^i)$ extracts all "positive" events from the input guard g^i (to be consumed when enabling the guard), i.e.,

$$e^c(\underline{e}_1\& \cdots \&\underline{e}_m\&\neg\underline{e}'_1\& \cdots \&\neg\underline{e}'_n) \ =_{df} \bigcup_{1\le i\le m} \underline{e}_i$$

and $e^g(g^o)$ records the set of events generated by the output guard g^o. Given an output guard $g^o = \rightarrow \underline{e}\&@(x = v)$, the generated events are

$$e^g(g^o) \ =_{df} \begin{cases} \underline{e} \cup \{\uparrow x\},\ \mathit{if}\ \sigma(x) < v, \\ \underline{e} \cup \{\downarrow x\},\ \mathit{if}\ \sigma(x) > v, \\ \underline{e},\ \mathit{otherwise}. \end{cases}$$

If no guard is enabled, the clock tick can be performed.

$$\frac{\forall k : 1 \le k \le n \bullet \neg(\sigma(b_k) \wedge (E \vdash g_k^i))}{\langle P, \sigma, E\rangle \xrightarrow{\surd} \langle P', \sigma, \emptyset\rangle}$$

where P' is the same as P if no time delay guards ($\#1$) appear in P. Otherwise, it is the guarded choice obtained from P by eliminating all time delay guards.

A parallel construct of guarded choices P is of the form $G_1 \parallel \cdots \parallel G_n$ where

$$G_k = [\!]_{1\le j\le r_k} b_{jk}\&g_{jk}^i\&g_{jk}^o\, P_{jk},\ 1 \le k \le n$$

This can be transformed into a guarded choice construct by algebraic laws ([7]). Here, we give the transition rules for the parallel construct directly. It can perform a (compositional) guarded transition if some threads agree, where $i_1, \cdots, i_n$ denotes a permutation of $1, \cdots, n$.

$$\frac{\begin{array}{c}\langle G_{i_k}, \sigma, E\rangle \xrightarrow{l_{i_k}} \langle P_{i_k}, \sigma_{i_k}, E_{i_k}\rangle,\ 1 \leq k \leq m \\ \forall j : 1 \leq j \leq r_k \bullet \neg(\sigma(b_{ji_k}) \wedge (E \vdash g^i_{ji_k})),\ m < k \leq n \\ \sigma' = \sigma_{i_1} \oplus \cdots \oplus \sigma_{i_m} \\ E' = (E - \bigcup_{1\leq k\leq m} e^c(g^i_{i_k})) \cup \bigcup_{1\leq k\leq m} e^g(g^o_{i_k})\end{array}}{\langle P, \sigma, E\rangle \stackrel{\&_{1\leq k\leq m} l_{i_k}}{\longrightarrow} \langle P', \sigma', E'\rangle}$$

where $P' \ =_{df} \ Q_1 \parallel \cdots \parallel Q_n$, and $Q_{i_k} =_{df} \begin{cases} P_{i_k},\ 1 \leq k \leq m, \\ G_{i_k},\ m < k \leq n \end{cases}$

If no threads can take a guarded transition, then the clock tick event can take place, as follows:

$$\frac{\forall j : 1 \leq j \leq r_k \bullet \neg(\sigma(b_{ji_k}) \wedge (E \vdash g^i_{ji_k})),\ 1 \leq k \leq n}{\langle P, \sigma, E\rangle \xrightarrow{\surd} \langle P', \sigma, \emptyset\rangle}$$

Note that P' is the same as P if no time delay guards ($\#1$) appear in P. Otherwise, it is the guarded choice obtained from P by eliminating all time delay guards.

A *sink* thread does not block the behaviour of its partners.

$$\frac{\langle P, \sigma, E\rangle \xrightarrow{l} \langle P', \sigma', E'\rangle}{\langle \mathit{sink} \parallel P, \sigma, E\rangle \xrightarrow{l} \langle \mathit{sink} \parallel P', \sigma', E'\rangle}$$

4 Mapping Statecharts into Verilog

In this section, we build a link between Statecharts and Verilog, by which a Statecharts description can be mapped to its corresponding Verilog program. We show such a mapping preserves the semantics and can be conducted in a compositional manner.

4.1 Mapping Function

Before constructing the mapping function called L, we address some subtle issues and introduce some notations. There exist two features which complicate the definition of L on an Or-chart, one is the hierarchical feature of Statecharts and the priority of transitions, whereas the other lies in that an And-chart can be a sub-chart of an Or-chart. This feature differentiates Statecharts from conventional programming languages. The former indicates that transitions in an outer level (rule **2**) has higher priority than those in an inner level (rule **3**). The possible transitions are considered hierarchically, starting from the current active state, and progressing into inner active substates where applicable. By enumerating these transitions in accordance with the hierarchy, we can cope with the different priorities for transitions occurring in distinct levels.

To deal with the above features, we prepare the following formal notations. We first give a function *or-depth* : $\mathsf{SC} \to \mathbb{N}$ to calculate the "or-depth" of a statechart, which is defined as follows:

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

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

Secondly, we extend some notations from Or-charts to And-charts. As already known, for an Or-chart $c = |[s : [p_1, \cdots, p_n], p_l, T]|$, *active*$(c) = p_l$ denotes its current active substate; for any transition $\tau \in T$, *src*(τ) and *tgt*(τ) respectively represent its source and target state. Given a parallel statechart $c = |[s : \{p_1, \cdots, p_n\}]|$, where all p_i are Or-charts, we define its current active state as a vector of the active states of these constituents, i.e., *active*$(c) =_{df} ($*active*$(p_1), \cdots,$ *active*$(p_n))$. We use $T(c)$ to denote all possible (perhaps compositional) transitions of the And-chart c. Given a transition $\tau = \&_{1 \le k \le m} \tau_{i_k} \in T(c)$, where $\tau_{i_k} \in T^*(p_{i_k})$, for $1 \le k \le m$, and $i_1, \cdots, i_n$ is a permutation of $1, \cdots, n$, we define its source state and target state respectively as follows:[2]

src$(\tau) =_{df} (q_1, \cdots, q_n)$, where $q_{i_k} =$ *src*(τ_{i_k}), for $1 \le k \le m$, and $q_{i_k} =$ *active*(p_{i_k}), for $m < k \le n$;

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

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 substates. We use a function

$$resc : \mathcal{T} \times \mathsf{SC} \to \mathsf{SC}$$

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-chart c, and any transition τ, *resc*$(\tau, c) =_{df} c$;
– for an Or-chart $c = |[s : [p_1, \cdots, p_n], p_l, T]|$, and a transition τ,
$$resc(\tau, c) =_{df} \begin{cases} c_{[l \mapsto \mathbf{a2d}(tgt(\tau))]}, & \textit{if } \tau \in T \wedge src(\tau) = p_l, \\ c_{[l \mapsto resc(\tau, p_l)]}, & \textit{if } \tau \in T^*(p_l), \\ c, & \textit{otherwise}. \end{cases}$$
– for an And-chart $c = |[s : \{p_1, \cdots, p_n\}]|$, and a transition τ,
$$resc(\tau, c) =_{df} \begin{cases} c_\tau, & \textit{if } \tau = \&_{1 \le k \le m} \tau_{i_k} \in T(c), \\ c, & \textit{otherwise}. \end{cases}$$
where $c_\tau = c[q_1/p_1, \cdots, q_n/p_n]$ is the statechart obtained from c via replacing p_i by q_i, for $1 \le i \le n$, $q_{i_k} =$ *resc*(τ_{i_k}, p_{i_k}), for $1 \le k \le m$, and $q_{i_k} = p_{i_k}$, for $m < k \le n$.

[2] For an Or-chart $p = |[s : [p_1, \cdots, p_n], p_l, T]|$, $T^*(p)$ contains all possible transitions inside p along its transitive active substate chain, i.e., $T^*(p) =_{df} \{\tau \mid \tau \in T \wedge src(\tau) = p_l\} \cup T^*(p_l)$.
With the help of $T^*(p)$, we define the aforementioned possible transition set $T(c)$ for an And-chart $c = |[s : \{p_1, p_2\}]|$ formally as $T(c) =_{df} \{\tau_i \& h_{3-i} \mid \tau_i \in T^*(p_i), i = 1, 2\} \cup \{\tau_1 \& \tau_2 \mid \tau_i \in T^*(p_i), i = 1, 2\}$, where $h_i =_{df} \&\{\neg\tau \mid \tau \in T^*(p_i)\}$. The transition set for the general And-chart with n components can be defined similarly.

The definition of L is split into three cases in accordance with the type of the source statechart.

Definition 1 (Mapping function L). *The function*

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

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

$\forall c \in \mathsf{SC} \bullet \textbf{\textit{vars}}(L(c)) = \textbf{\textit{vars}}(c)$

and it is inductively defined as follows.

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

  $L(c) \ =_{df} \ sink$

- *For a statechart $c = |[s : \{p_1, \cdots, p_n\}]|$ constructed by* And*, L maps it into a parallel construct in Verilog.*

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

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

  $$L(c) \ =_{df} \ \begin{cases} sink, \ \mathit{if}\, T^*(c) = \emptyset \\ P, \ otherwise \end{cases}$$

  where

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

  and

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

  The input guard $g^i_{\tau_k}$ comprises the overall trigger events of τ_k, which has the form $\underline{e}_1 \& \neg \underline{e}_2$, where $\underline{e}_1$ are events from $\mathit{trig}^+(\tau_k)$, whereas $\underline{e}_2$ are events out of $\mathit{trig}^-(\tau_k)$.
  Due to the priority mechanism of Statecharts, an enabled transition τ_k in an inner level (k) can occur only when no transitions from any outer level $(0, \cdots, k\text{-}1)$ are enabled. The part $(\&_{1 \le j \le k} \, h_j)$ is used to denote this condition.
  The output guard $g^o_{\tau_k}$ is the overall action performed by τ_k, which has the form $\rightarrow \underline{e} \& @(x = v)$, where $\underline{e}$ comprises all abstract events out of $a^e(\tau_k)$, and the assignment action $x = v$ is from $a^a(\tau_k)$.

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*.

The following example deals with the transformation of statecharts in Fig. 2.

Example 1. The statechart (a) in Fig. 2 can be described as p:

$$p = |[s : [p_1, p_2], p_1, \{\tau_1, \tau_2\}]|$$

where $\tau_i \; =_{df} \; \langle p_i, \{e\}, \emptyset, true, p_{3-i}\rangle, \; i = 1, 2$.

After applying the mapping function L onto it, the statechart (a) becomes the following process

$$\mu X \bullet (e \; (e \; X))$$

which does nothing but just waits to be fired by an event e from the environment.

The statechart (b) can be described as q:

$$q = |[s : \{q_1, q_2\}]|$$
$$q_1 = |[s_1 : [p_3, p_4], p_3, \{\tau_3\}]|$$
$$q_2 = |[s_2 : [p_5, p_6, p_7], p_5, \{\tau_4, \tau_5\}]|$$

where

$$\tau_3 = \langle p_3, \{a, b\}, \{e\}, true, p_4\rangle$$
$$\tau_4 = \langle p_5, \{b, c\}, \{f\}, true, p_6\rangle$$
$$\tau_5 = \langle p_5, \{e\}, \{g\}, true, p_7\rangle$$

It is mapped into the following parallel construct

$$(a\&b\&(\rightarrow e) \; sink) \parallel ((e\&(\rightarrow g) \; sink)[\!]((b\&c)\&(\rightarrow f) \; sink))$$

where the two parallel processes are mapped from q_1 and q_2, respectively. □

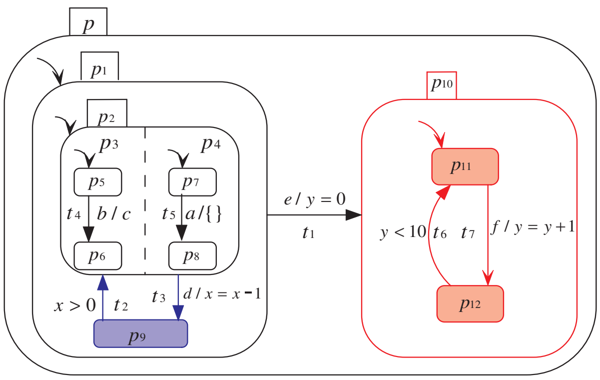

Fig. 3. A More Complicated Statechart

Example 2. The statechart in Fig. 3 is more complicated than those in Fig. 2. It is described by:

$$p = |[s : [p_1, p_{10}], p_1, \{t_1\}]|$$
$$p_1 = |[s_1 : [p_2, p_9], p_2, \{t_2, t_3\}]|$$
$$p_2 = |[s_2 : \{p_3, p_4\}]|$$
$$p_3 = |[s_3 : [p_5, p_6], p_5, \{t_4\}]|$$
$$p_4 = |[s_4 : [p_7, p_8], p_7, \{t_5\}]|$$
$$p_{10} = |[s_{10} : [p_{11}, p_{12}], p_{11}, \{t_6, t_7\}]|$$

where

$$\begin{aligned}
t_1 &= \langle p_1, \{e\}, \{@(y=0)\}, true, p_{10}\rangle \\
t_2 &= \langle p_9, \emptyset, \emptyset, x > 0, p_2\rangle \\
t_3 &= \langle p_2, \{d\}, \{@(x=x-1)\}, true, p_9\rangle \\
t_4 &= \langle p_5, \{b\}, \{c\}, true, p_6\rangle \\
t_5 &= \langle p_7, \{a\}, \emptyset, true, p_8\rangle \\
t_6 &= \langle p_{12}, \emptyset, \emptyset, y < 10, p_{11}\rangle \\
t_7 &= \langle p_{11}, \{f\}, \{@(y=y+1)\}, true, p_{12}\rangle
\end{aligned}$$

After applying L onto it, we obtain the following recursive process.

$$\mu X \bullet \begin{pmatrix} Q \parallel P \parallel (b\&\neg a\&\neg d\&\neg e\& \to c)\ (Q \parallel P \parallel (a\&\neg d\&\neg e)\ (Q \parallel P)) \\ \parallel (a\&\neg b\&\neg d\&\neg e)\ (Q \parallel P \parallel (b\&\neg d\&\neg e\& \to c)\ (Q \parallel P)) \\ \parallel (b\&a\&\neg d\&\neg e\& \to c)\ (Q \parallel P) \end{pmatrix}$$

where $Q =_{df}\ e\&@(y=0)\ \mu Y \bullet (f\&@(y=y+1)\ (y<10)\ Y)$
$P =_{df}\ (d\&\neg e\&@(x=x-1))\ ((x>0)\&\neg e)\ X$

Let us illustrate a more practical example: a simple remote controller for an air-conditioner.

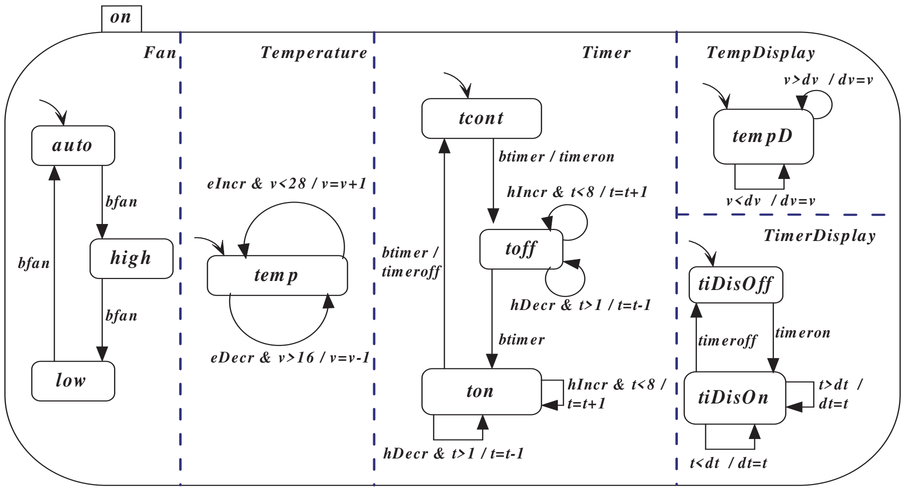

Fig. 4. An Air-Conditioner Remote Controller: the *on* state

Example 3. Part of the specification for an air-conditioner remote controller is presented in Fig. 4. It is composed of five orthogonal components namely *Fan*, *Temperature*, *Timer*, *TempDisplay*, and *TimerDisplay*. They will be respectively mapped to Verilog programs *pFan*, *pTemperature*, *pTimer*, *pTempDisplay*, and *pTimerDisplay*.

After applying the mapping function L to the statechart in Fig.4, we obtain the following target program *pon*:

$$pon\ =_{df}\ pFan \parallel pTemperature \parallel pTimer \parallel pTempDisplay \parallel pTimerDisplay$$

The five component programs are respectively

$$pFan \ =_{df} \ \mu X \bullet (bfan\ (bfan\ (bfan\ X)))$$

$$pTemperature \ =_{df} \ \mu X \bullet \left(\begin{array}{l} ((v < 28)\&eIncr\&@(v = v + 1)\ X) \\ \talloblong\ ((v > 16)\&eDecr\&@(v = v - 1)\ X) \end{array} \right)$$

$$pTimer \ =_{df} \ \mu X \bullet ((btimer\&\ \rightarrow timeron)\ P)$$

where

$$P \ =_{df} \ \mu Y \bullet \left(\begin{array}{l} ((t < 8)\&hIncr\&@(t = t + 1)\ Y) \\ \talloblong\ ((t > 1)\&hDecr\&@(t = t - 1)\ Y) \\ \talloblong\ (btimer\ Q) \end{array} \right)$$

$$Q \ =_{df} \ \mu Z \bullet \left(\begin{array}{l} ((t < 8)\&hIncr\&@(t = t + 1)\ Z) \\ \talloblong\ ((t > 1)\&hDecr\&@(t = t - 1)\ Z) \\ \talloblong\ ((btimer\&\ \rightarrow timeroff)\ X) \end{array} \right)$$

$$pTempDisplay \ =_{df} \ \mu X \bullet \left(\begin{array}{l} ((v > dv)\&@(dv = v)\ X) \\ \talloblong\ ((v < dv)\&@(dv = v)\ X) \end{array} \right)$$

$$pTimerDisplay \ =_{df} \ \mu X \bullet timeron\ \mu Y \bullet \left(\begin{array}{l} ((t > dt)\&@(dt = t)\ Y) \\ \talloblong\ ((t < dt)\&@(dt = t)\ Y) \\ \talloblong\ (timeroff\ X) \end{array} \right)$$

4.2 Correctness

The following theorem shows that the mapping function from Statecharts into Verilog is a homomorphism between the two formalisms.

Theorem 1 (Homomorphism). *Given any statechart C and any of its possible transitions τ which leads to statechart C', there exists a Verilog transition l for $L(C)$ which arrives at P', such that $P' = L(C')$; on the other hand, for any Verilog transition of $L(C)$ leading to P', there exists a Statecharts transition from C to C', such that $L(C') = P'$, as illustrated in Fig. 5.*

$$\begin{tikzcd} C \arrow[r, "t"] \arrow[d, "L"'] & C' \arrow[d, "L"] \\ P \arrow[r, "l"'] & P' \end{tikzcd}$$

Fig. 5. Mapping function L

Proof By case analysis on the type of C.

1. $C = \|[s]\|$ is constructed by Basic.
   What C can do is to perform the clock tick and remains as C after the transition. On the other hand, from Definition 1 we know $L(C) = \mathit{sink}$, which does nothing but performs the clock tick and remains as *sink* after that.
2. $C = \|[s : [p_1, \cdots, p_n], p_l, T]\|$ is constructed by Or.
   In case that $T^*(C) = \emptyset$, it can be proved similar to the first case. Now suppose $T^*(C) \neq \emptyset$, C can (1) perform a transition $\tau \in T(\mathit{active}^k(C))$ for some $k \geq 0$ in case that all transitions of outer levels (if any) are not available, which changes the active substate of $\mathit{active}^k(C)$ from its source state to its target state and results in $\mathit{resc}(\tau, C)$; (2) otherwise, it can take a clock tick and remain its state. From Definition 1 of L, we know that $L(C)$ has the form $[\!]\,(g_\tau\, P_\tau)$. If (1) occurs, g_τ is fired, from the semantics of Verilog, such a program can perform the corresponding transition and become P_τ, otherwise it can perform the clock tick transition. From the definition of L, it is straightforward that $P_\tau = L(\mathit{resc}(\tau, C))$.
   The second part can be proved similarly from the definition of L.
3. $C = \|[s : \{p_1, \cdots, p_n\}]\|$ is constructed by And.
   From Definition 1, we know

   $$L(C) = L(p_1) \parallel \cdots \parallel L(p_n).$$

   Given any possible transition $\tau \in T(C)$, we assume $\tau = \&_{1 \leq k \leq m}\, \tau_k$, where $\tau_k \in T^*(p_k)$, without loss of generality. If τ can be performed at the current environment, from rule **5**, we know that τ_k, for $1 \leq k \leq m$, are ready to take place and orthogonal components other than $p_1, \cdots, p_m$ do not have available transitions. This implies all processes $L(p_1), \cdots, L(p_m)$ can take the transition corresponding to $\tau_1, \cdots, \tau_m$ respectively in the current environment, whereas others can not. From the operational semantics of parallel construct of Verilog, a parallel transition corresponding to τ can take place and after the transition the program becomes

   $$P_1 \parallel \cdots \parallel P_n$$

   where

   $$P_i = \begin{cases} L(\mathit{resc}(\tau_i, p_i)), \ \mathit{for}\ 1 \leq i \leq m, \\ L(p_i), \ \mathit{otherwise.} \end{cases}$$

   It exactly accords with $L(\mathit{resc}(\tau, C))$. The case for a clock tick transition is trivial. The second part is also straightforward, since any transition of the result parallel construct $L(C)$ in Verilog either involves several threads or a single thread. From the definition of L, we can conclude, in either case, there exists a corresponding Statecharts transition for C, which yields C' and $L(C') = P'$ holds. □

The following theorem shows the soundness of the mapping function.

Theorem 2 (Soundness). *The mapping function L in Definition 1 transforms any Statecharts specification into a Verilog program with the same observable behaviour as the original chart.*

Proof In addition to the results from Theorem 1, we need to show that, given a statechart C and its image $L(C)$ in Verilog, any possible pair of their corresponding steps (a statechart transition and a Verilog transition), starting from the same context environment(the same σ and E in the corresponding configurations), consume the same set of events, generate the same set of events, and bring the updates of data state into accord. These follow directly from the construction of the mapping function L. $\square$

5 Discussion and Related Work

In our co-specification process, we conduct the partitioning task after a Verilog behaviour specification has been generated from the higher level system description in Statecharts. We use this approach because the semantics of Verilog has been well investigated and a collection of algebraic laws ([7]) can be used as the fundamental support of the partitioning algorithm. In contrast, most work on Statecharts' semantics focuses on its operational rules since it has proved to be quite difficult to present a simple denotational model from which algebraic laws of Statecharts can be derived. Due to this difficulty, the partitioning problem is currently not addressed at the Statecharts level. Although it may seem unnatural to obtain a software specification in Verilog after partitioning, it is still reasonable in the sense that the behaviour subset of Verilog is very similar to C programming language and can be readily transformed into C code.

Due to the involvedness of formal semantics for Statecharts, there have been so many related works that we can hardly discuss all here. Some of them are presented in [6, 9, 11, 12, 14, 21]. Many of these works adopt the simpler synchronous model. The work in [6] takes into account a very large subset of Statecharts, but the semantics is neither compositional nor formal. In contrast, our operational semantics is formal, compositional and supports asynchronous model.

Although it is reported that Verilog has been widely used in industry (especially in United States) for years, its precise semantics has been ignored until recently. The results [8, 22, 23, 7] are all based on Gordon's interpretation on simulation cycles [3]. A simple operational semantics is given in [8]. Zhu, Bowen and He [22, 23] investigate the consistency between Verilog's operational and denotational semantics, while He [7] explores a program algebra for Verilog and its connection with both operational and denotational semantics.

Some of related works on connecting Statecharts with other formalisms are presented in [1, 2, 13, 19, 20, 18]. Beauvais et.al. [1] and Seshia et.al. [19] 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 [6] lacking correctness proofs. The authors of [2, 13] transform variants of Statecharts into hierarchical timed automata and use tools (UPPAAL, SPIN) to model check Statecharts properties. Also, [20] based on the denotational semantics [9] aims to connect a subset of Statecharts with temporal logic *FNLOG* for theoretically proving Statecharts' properties. More recently, a translation from Statecharts to B/AMN is reported in [18]. 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. Our mapping function is constructed based on formal semantics for both the source and target formalisms and has been proven to be semantics-preserving.

6 Conclusion

This paper proposes a mapping function which transforms a high level specification in visual formalism Statecharts into a behaviour description in Verilog HDL. We explore a compositional operational semantics to Statecharts which contains many powerful features that Statecharts owns, but proved to be difficult to be combined into a uniform formalism. Based on this semantics and an operational semantics for Verilog, we show our mapping function provides as a semantic link between the two formalisms. Moreover, we combine this transformation process with our previous formal partitioning approach yielding a hardware/software co-specification process that can be automated. However, the translation from Statecharts to Verilog can also be used in pure hardware design. After translating into a behaviourial description in Verilog, existed Verilog synthesizer can be used to obtain low level descriptions, like netlists, for direct implementation in hardware (ASICs or FPGAs).

As an immediate future work, the obtained guarded choice specification should be transformed into simplified behaviourial description in Verilog using algebraic laws [7]. An implementation for this mapping from graphical descriptions in Statecharts to Verilog specifications is also being considered.

Acknowledgement

We would like to thank Jifeng He for inspiration, thank Khoo Siau Cheng, P.S. Thiagarajan, Wang Yi and Zhu Huibiao for useful discussions. We are also grateful to anonymous referees for many helpful comments.

References

1. J.-R. Beauvais, et. al., "A Translation of Statecharts to Signal/DC+", Technical Report, IRISA, 1997.
2. Alexandre David, M. Oliver Möller and Wang Yi, "Formal Verification of UML Statecharts with Real-Time Extensions", in the *Proc. of Fundamental Approaches to Software Engineering (FASE 2002)*, LNCS 2306, pp. 218–232, Springer-Verlag, 2002.
3. M. Gordon, "The Semantic Challenge of Verilog HDL", In *the Proc. of Tenth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 136–145, 1995.
4. D. Harel, "Statecharts: a Visual Formalism for Complex Systems", *Science of Computer Programming*, vol.8, no.3, pp. 231–274, 1987.
5. D. Harel, "On Visual Formalisms", *Communications of the ACM*, Vol. 31, No. 5, pp. 541–530, 1988.
6. 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.
7. J. He, "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)*, Springer-Verlag, 2002.

8. J. He and Q. Xu, “An Operational Semantics of a Simulator Algorithm”, in the *Proc. of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’2000)*, Las Vegas, Nevada, USA, June 26-29, 2000.
9. J.J.M. Hooman, S. Ramesh, and W.P. de Roever, “A Compositional Axiomatization of Statecharts”, *Theoretical Computer Science* 101, pp. 289–335, 1992.
10. IEEE Computer Society, *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE std 1364-1995)*, 1995.
11. G. Lüttgen, M. von der Beeck, and R. Cleaveland, “A Compositional Approach to Statecharts Semantics”, NASA/CR-2000-210086, ICASE Report No.2000-12, March, 2000.
12. A. Maggiolo-Schettini, A. Peron, and S. Tini, “Equivalences of Statecharts”, in *7th International Conference on Concurrency Theory (CONCUR’96)*, Pisa, Italy, Aug. 1996, LNCS 1119, pp.687–702, Springer-Verlag.
13. E. Mikk, Y. Lakhnech, M. Siegel and G. Holzmann, “Implementing Statecharts in Promela/SPIN”, in the *Proc. of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, IEEE Computer Society, 1999.
14. 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, Springer-Verlag, Berlin.
15. S. Qin and J. He, “An Algebraic Approach to Hardware/software Partitioning”, in *the Proc of the 7th IEEE International Conference on Electronics, Circuits and Systems (ICECS’2k)*, IEEE Computer Society Press, pp 273–276, Lebanon, Dec., 2000.
16. S. Qin and J. He, “Partitioning Program into Hardware and Software”, in *the Proc of APSEC 2001*, IEEE Computer Society Press, pp. 309–316, Macau, Dec., 2001.
17. S. Qin, J. He, Z. Qiu, and N. Zhang, “Hardware/Software Partitioning in Verilog”, in the *4th International Conference for Formal Engineering Methods (ICFEM2002)*, LNCS 2495, pp. 168–179, Springer-Verlag.
18. 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. 128–144, Springer-Verlag, 2002.
19. 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.
20. A. Sowmya and S. Ramesh, “Extending Statecharts with Temporal Logic”, *IEEE Transactions on Software Engineering*, Vol. 24, No. 3, March, 1998.
21. M. von der Beeck, “A Comparison of Statecharts Variants”, in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, L. de Roever and J. Vytopil, Eds. LNCS 863, pp. 128–148, Springer-Verlag, New York.
22. Zhu H., J. Bowen and He J., “From Operational Semantics to Denotational Semantics for Verilog”, in *the Proc. of CHARME 2001*, *LNCS* 2144, pp. 449–464.
23. H. Zhu, J. Bowen and J. He, “Soundness, Completeness and Non-redundancy of Operational Semantics for Verilog Based on Denotational Semantics”, in the *4th International Conference for Formal Engineering Methods (ICFEM2002)*, LNCS 2495, pp. 600–612, Springer-Verlag.