Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

Shengchao Qin · Wei-Ngan Chin · Jifeng He · Zongyan Qiu

From Statecharts to Verilog: a Formal Approach to Hardware/Software Co-Specification

Received: date / Accepted: date

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 first investigate the Statecharts formalism by providing it a formal syntax and a compositional operational semantics. Based on that, a semantics-preserving linking function is designed to compile specifications written in Statecharts into Verilog. The obtained Verilog specifications are then passed to a partitioning process to generate hardware and software sub-specifications, where the correctness is guaranteed by algebraic laws of Verilog.

Keywords Statecharts · Verilog · operational semantics · homomorphism · algebraic laws · hardware/software partitioning

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 behaviors 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 formula 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. In our previous work [23; 24], we propose a formal partitioning algorithm which splits an Occam source program into hardware and software specifications. The partitioning correctness is verified using algebraic laws developed for Occam. One of advantages of this approach is that it ensures the correctness of the partitioning process. Moreover, it optimizes the underlying target architecture, and facilitates the reuse of hardware devices.

In this paper, we first bridge the gap between the high level specification in Statecharts and the Verilog source program by defining a mapping function between

Part of the work appeared earlier in FME03 and ICFEM02. Wei-Ngan Chin is supported by Singapore research grant R-252-000-151-112. Jifeng He is supported by China 973 project 2002CB312001. Zongyan Qiu is supported by NNSFC 60173003 and 60573081.

Shengchao Qin (Correspondence)
Dept of Computer Science, Durham University
South Road, Durham, DH1 3LE, UK
Tel: +44 191 334 1745 Fax: +44 191 334 1701
E-mail: shengchao.qin@durham.ac.uk

Wei-Ngan Chin
Dept of Computer Science, National University of Singapore
E-mail: chinwn@comp.nus.edu.sg

Jifeng He
Software Engineering Inst., East China Normal University
E-mail: jifeng@sei.ecnu.edu.cn

Zongyan Qiu
School of Mathematical Sciences, Peking University
E-mail: zyqiu@pku.edu.cn

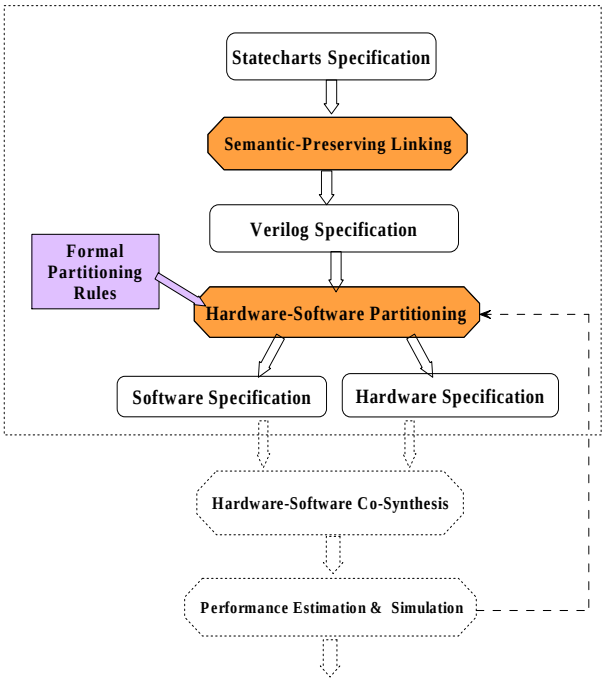

Fig. 1 HW-SW Co-Specification Process

the two formalisms; and then present an algebraic hardware/software partitioning process in Verilog. The overall co-specification process can thus be automated, as illustrated in Fig.1. We have made the following key contributions:

- 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 specification in Statecharts into a program in Verilog. We show that the target program after mapping preserves the semantics of the source specification. This mapping is taken as the front-end of our hardware/software co-specification process; and
- we design a collection of formal rules to partition the above-obtained Verilog specification into hardware and software sub-specifications. All the rules are proved correct based on the algebraic semantics of Verilog.

The remainder of this paper is organized as follows. Section 2 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. An operational semantics and an algebraic semantics are presented. We build a mapping function from Statecharts into Verilog and prove that it is a homomorphism between the two formalisms in Section 4. Section 5 presents our hardware/software partitioning process, where the partitioning strategy and formal partitioning rules are depicted. Related work together with a simple conclusion follow afterwards.

2 Operational Semantics for Statecharts

The graphical language of Statecharts as proposed by David Harel [5] 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 [22], 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 [30], each of which can be regarded as a subset of the originally expected language. The version discussed in [7] for STATEMATE is rather large and powerful; however, their operational semantics is neither formal nor compositional. The work presented in [18] provides a compositional semantics for Statecharts, but does not contain data states. Hooman *et.al* [15] propose a denotational semantics based on histories of computation. Following this line, [29] 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 formalize, 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]
- It covers the data-state issues of Statecharts, allowing assignments in state transitions.

[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 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 [6], *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, output events in Statecharts are asynchronous, and can be broadcast to any receiver without waiting. However, input events in Statecharts are synchronous, and are blocked until the arrival of the corresponding output events. Such a communication mechanism is similar to Verilog.

In order to formalize 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 [19; 18], 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.
– $\mathsf{Or}(s, [p_1, .., p_l, .., p_n], p_l, T)$ 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.
– $\mathsf{And}(s, \{p_1, .., p_n\})$ 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.

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

where

$src(\tau)$ and $tgt(\tau)$ 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 $\sigma' = \sigma \oplus \{v \mapsto \sigma(ex)\}$.

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

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

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.

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

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 [22] 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. □

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**.

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

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

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

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 an AND-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 charts 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 $1, \cdots, n$.

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

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.

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

3 Verilog and Its Formal 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 ([16]) are two contemporary HDLs that have been used for years, where Verilog HDL has been standardized and widely adopted in industry [16]. Verilog programs can exhibit a rich variety of behaviours, including event-driven computation and shared-variable concurrency. In our hardware/software partitioning process, the non-trivial subset of Verilog we adopt contains the following categories of syntactic elements.

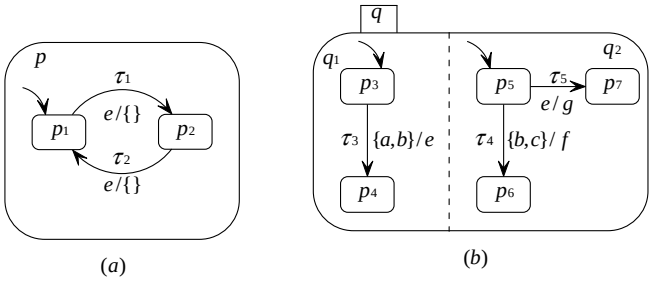

Fig. 2 Example statecharts (a) and (b)

1. A Verilog program can be a sequential process or a program paralleled by several sequential processes, with or without local variable declaration.

$P ::= S \mid P \parallel P \mid var\ x \bullet P$

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

$S ::=$ PC(primitive command)
| $S; S$ (sequential composition)
| $S \lhd b \rhd S$ (conditional)
| $b * S$ (iteration)
| $(g\ S) [\!] \ldots [\!] (g\ S)$ (guarded choice)
| $always\ S$ (infinite loop)
| $case(e)\ (pt\ S) \ldots (pt\ S)$ (switch statement)

where

$PC ::= v := e \mid sink \mid skip \mid \bot \mid \rightarrow \eta \mid v := cg\ e$
$g ::= \#n$ (time delay) $\mid eg$ (event control)
$\mid \rightarrow \eta$ (output event)
$cg ::= \#n \mid eg$
$eg ::= \eta \mid eg\ or\ eg \mid eg\ \&\ eg \mid eg\ \&\ \neg eg$
$\eta ::= \sim v$ (value change) $\mid \uparrow v$ (value rising)
$\mid \downarrow v$ (value falling) $\mid \underline{e}$ (a set of abstract events)

To facilitate algebraic reasoning, the language is enriched with

- assignment event $@(v := e)$
- general guarded choice construct $(g_1\ P_1) [\!] \ldots [\!] (g_n\ P_n)$
- non-deterministic choice $P \sqcap Q$

Although it is reported that Verilog has been much more widely used in industry than VHDL [3], the formal semantics of Verilog has not been fully studied. Gordon [4] tries to relate event semantics of Verilog to its trace semantics. He and Zhu [13; 35] explore an operational and a denotational semantics for Verilog and investigate some algebraic laws from them. Zhu, Bowen and He [32; 34; 33] establish formal consistency between abovementioned two presentations. Iyoda and He [17] successfully apply simple algebraic laws of Verilog to hardware synthesis process. In [8], He has explored a collection of algebraic laws for Verilog, by which a well-formed Verilog program can be transformed into head normal forms. In the following, we first present an operational semantics for the subset of Verilog that we adopt and then explore some algebraic laws for it. The operational semantics will be used to build the formal link between Statecharts and Verilog, while the algebraic semantics will play a fundamental role in our hardware/software partitioning process. The discussion on the consistency between the operational and algebraic semantics is out of the scope of this paper, readers can refer to [31] where a detailed discussion on unifying different semantics is available.

3.1 Operational Semantics

The subset of Verilog we adopt is quite similar to that proposed by He [8]. However, there are some different treatments between our version and that in [8]. We include explicitly the possible context environment of active events in our configuration, and change the operational rules for the parallel constructs. This facilitates the 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.
- $\sigma : Var \rightarrow Val$ 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.

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

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.

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

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

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

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 [8]. 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 $1, \cdots, n$.

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

where $P' =_{df} Q_1 \parallel \cdots \parallel Q_n$, and

$$Q_{i_k} =_{df} \begin{cases} P_{i_k}, & 1 \le k \le m, \\ G_{i_k}, & m < k \le n \end{cases}$$

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

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

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

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

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

3.2 Algebraic Laws

We discuss the algebraic semantics of Verilog in this section, which will be useful in later discussions. Before presenting the algebraic laws, we define a triggering predicate as follows.

Definition 1 Given an event control eg, we define those simple events that enable eg as follows.

$$tr(eg) =_{df} \left\{ \begin{array}{ll} \{\uparrow x\}, & \text{if } eg = \uparrow x \\ \{\downarrow x\}, & \text{if } eg = \downarrow x \\ \{\uparrow x, \downarrow x\}, & \text{if } eg = \sim x \\ tr(eg_1) \cup tr(eg_2), & \text{if } eg = eg_1 \textit{ or } eg_2 \\ tr(eg_1) \cap tr(eg_2), & \text{if } eg = eg_1 \ \&\ eg_2 \\ tr(eg_1) \setminus tr(eg_2), & \text{if } eg = eg_1 \ \&\ \neg eg_2 \end{array} \right\}$$

Given an output event $\rightarrow \eta$, and an event control eg, we adopt a triggering predicate, denoted as $\eta \rightsquigarrow eg$, to describe the condition under which the former enables the later.

$$\eta \rightsquigarrow eg =_{df} tr(\eta) \subseteq tr(eg)$$

and adopt the predicate, $\eta \not\rightsquigarrow eg$, to denote the condition when the former cannot trigger the later.

$$\eta \not\rightsquigarrow eg =_{df} tr(\eta) \cap tr(eg) = \emptyset$$

□

By this definition, we can define the well-formedness of guarded choice constructs:

Definition 2 A guarded choice $[\!]_{i \in I}\ g_i\ P_i$ is well-formed if and only if all its input guards are disjoint, i.e., for any input guards g_k, g_l from $\{g_i \mid i \in I\}$, if $tr(g_k) \cap tr(g_l) \neq \emptyset$, then $g_k = g_l$, and P_k and P_l are exactly the same process. □

All guarded choice constructs are well-formed in later discussions.

Now, we explore a collection of useful algebraic laws for Verilog programs.

Successive assignments to the same variable can be combined to a single one.
(*assgn-1*) $v := e;\ v := f \ = \ v := f[e/v]$

In an assignment to a list of variables, the order of variables is irrelative.
(*assgn-2*) $u, v := e, f \ = \ v, u := f, e$

Variables not occurred on the left side of an assignment remain unchanged during the assignment.
(*assgn-3*) $u := e \ = \ u, v := e, v$

skip does not change the value of any variable.
(*assgn-4*) $skip \ = \ v := v$

Sequential composition is associative, and has left zero $\perp$. It distributes backward over conditional, internal and external choices.
(*seq-1*) $(P; Q); R \ = \ P; (Q; R)$

(*seq-2*) $\perp; P \ = \ \perp$

(*seq-3*) $(P \sqcap Q); R \ = \ (P; R) \sqcap (Q; R)$

(*seq-4*) $(P \lhd b \rhd Q); R \ = \ (P; R) \lhd b \rhd (Q; R)$

(*seq-5*) $([\!]_{i\in I}\,(g_i\, Q_i)); R \ = \ [\!]_{i\in I}\,(g_i\,(Q_i; R))$

By the following law, we can transform a sequential composition of an output event and a guarded choice into a guarded process $(g\, P)$, where output guard g will no longer fire guards of P.
(*seq-6*) Let $S = [\!]_{i\in I}\,(g_i\, P_i)$, and g is the disjunction of all input guards of S.

(1). $\rightarrow\eta; S \ = \ \begin{cases} \rightarrow\eta\, S & \text{if } \eta \rightsquigarrow g; \\ \rightarrow\eta\, P_k & \text{if } \eta \rightsquigarrow g_k \text{ for some } k \in I. \end{cases}$

(2). $(x < f)_\perp; @(x := f); S \ =$
$\begin{cases} (x < f)_\perp; @(x := f)\, S & \text{if } \uparrow x \rightsquigarrow g; \\ (x < f)_\perp; @(x := f)\, P_k & \text{if } \uparrow x \rightsquigarrow g_k \text{ for some } k \in I. \end{cases}$

(3). $(x > f)_\perp; @(x := f); S \ =$
$\begin{cases} (x > f)_\perp; @(x := f)\, S & \text{if } \downarrow x \rightsquigarrow g; \\ (x > f)_\perp; @(x := f)\, P_k & \text{if } \downarrow x \rightsquigarrow g_k \text{ for some } k \in I. \end{cases}$

(4). $(x = f)_\perp; @(x := f); S \ = \ (x = f)_\perp; @(x := x)\, S$

where $b_\perp$ is an assertion defined as $skip \lhd b \rhd \perp$ ([14]).

For a general guarded choice G, we can also transform it by this law into a guarded choice $[\!]_{i\in I}\,(g_i\, P_i)$, where no output guard in $\{g_i \mid i \in I\}$ will enable any guards of the process following it. Without loss of generality, from now on, we assume all guarded choices meet this property.

Assignment distributes forward over conditional.

(*cond-1*) $v := e; (P \lhd b(v) \rhd Q) \ = \ (v := e; P) \lhd b(e) \rhd (v := e; Q)$

Iteration is subject to the fixed point theorem.
(*iter-1*) $b * P \ = \ (P; b * P) \lhd b \rhd skip$

Non-deterministic choice is idempotent, symmetric and associative.
(*nond-1*) $P \sqcap P \ = \ P$

(*nond-2*) $P \sqcap Q \ = \ Q \sqcap P$

(*nond-3*) $P \sqcap (Q \sqcap R) \ = \ (P \sqcap Q) \sqcap R$

Parallel operator is symmetric and associative, and has $\perp$ as zero.
(*par-1*) $P \parallel Q \ = \ Q \parallel P$

(*par-2*) $P \parallel (Q \parallel R) \ = \ (P \parallel Q) \parallel R$

(*par-3*) $\perp \parallel P \ = \ \perp$

Local variable declaration enjoys the following laws.
(*lvar-1*) $var\, x \bullet (x := e) \ = \ skip$

(*lvar-2*) $var\, x \bullet (P \lhd b \rhd Q) \ = \ (var\, x \bullet P) \lhd b \rhd (var\, x \bullet Q)$, provided x is not free in b.

(*lvar-3*) If x is not free in Q, then
(1) $var\, x \bullet Q \ = \ Q$
(2) $(var\, x \bullet P); Q \ = \ var\, x \bullet (P; Q)$
(3) $Q; (var\, x \bullet P) \ = \ var\, x \bullet (Q; P)$
(4) $(var\, x \bullet P) \parallel Q \ = \ var\, x \bullet (P \parallel Q)$

(*lvar-4*) $var\, v \bullet (\rightarrow\eta_v\, P) \ = \ var\, v \bullet (skip; P)$

(*lvar-5*) $var\, u \bullet (var\, v \bullet P) \ = \ var\, v \bullet (var\, u \bullet P)$

We will denote $var\, x \bullet var\, y \bullet \ldots \bullet var\, z$ as $var\, x, y, \ldots, z$.

The following is a set of expansion laws which enables us to convert a parallel process into a guarded choice. We assume that

$$G_1 \ = \ [\!]_{i\in I}(g_i\, Q_i) \qquad G_2 \ = \ [\!]_{j\in J}(h_j\, R_j)$$

$$G_3 \ = \ [\!]_{k\in K}(e_{v_k}\, P_k) \qquad G_4 \ = \ [\!]_{l\in L}(e_{u_l}\, T_l)$$

where all g_i and h_j are input guards (like η); all e_{v_k} and e_{u_l} are respectively output events with respect to variables v_k and u_l (like $\rightarrow\eta$ or $@(x := f)$).

(*par-4*) $(x := e; G_1) \parallel (y := f; G_2) \ =$
$(@(x := e)\,(G_1 \parallel (y := f; G_2))) \,[\!]$
$(@(y := f)\,((x := e; G_1) \parallel G_2))$

(*par-5*) $G_1 \parallel (y := f; G_2) \ =$
$(@(y := f)(G_1 \parallel G_2)) \,[\!]$
$[\!]_{i\in I}\, g_i\,(Q_i \parallel (y := f; G_2))$

(*par-6*) Let $g =_{df} or_{i\in I}\, g_i$, $h =_{df} or_{j\in J}\, h_j$, then
$(G_1 \,[\!]\, G_3) \parallel (G_2 \,[\!]\, G_4) \ =$
$[\!]_{i\in I}\,((g_i \,\&\, \neg h)\,(Q_i \parallel (G_2 \,[\!]\, G_4)) \,[\!]$

$$\begin{array}{l}
\|_{j\in J}\ ((h_j \,\&\, \neg g)\,((G_1 \| G_3) \parallel R_j))\ \| \\
\|_{i\in I, j\in J}\ ((g_i \,\&\, h_j)\,(Q_i \parallel R_j))\ \| \\
\|_{k\in K, j\in J, e_{v_k}\rightsquigarrow h_j}\ (e_{v_k}\,(P_k \parallel R_j))\ \| \\
\|_{k\in K, e_{v_k}\rightsquigarrow h}\ (e_{v_k}\,(P_k \parallel (G_2 \| G_4)))\ \| \\
\|_{i\in I, l\in L, e_{u_l}\rightsquigarrow g_i}\ (e_{u_l}\,(Q_i \parallel T_l))\ \| \\
\|_{l\in L, e_{u_l}\rightsquigarrow g}\ (e_{u_l}\,((G_1 \| G_3) \parallel T_l))
\end{array}$$

(*par-7*) An assignment thread is involved.

(1) $(x := e) \parallel (y := f) \;=\; (@(x := e)\,(y := f)) \,\|\, (@(y := f)\,(x := e))$

(2) $(x := e) \parallel G_2 \;=\; (@(x := e)\,G_2) \,\|\, \|_{j\in J}\,(h_j\,((x := e) \parallel R_j))$

The parallel operator is disjunctive.

(*par-8*) $(P \sqcap Q) \parallel R \;=\; (P \parallel R) \sqcap (Q \parallel R)$

In some special case, the parallel operator distributes over conditional.

(*par-9*) $var\, v_1, \ldots, v_n \bullet ((S_1 \lhd b \rhd S_2) \parallel G) \;=\; var\, v_1, \ldots, v_n \bullet ((S_1 \parallel G) \lhd b \rhd (S_2 \parallel G))$,

provided guards in G are either event controls with respect to variables in $\{v_1, \ldots, v_n\}$ or time-delay guards.

Time-delay guards are involved in the following law.

(*par-10*) Let $n_1 > n_2 > 0$, $n > 0$.

(1). $(\#n\,S) \| G_3 \;=\; G_3$

(2). $$\begin{array}{l}
(G_1 \| \#n_1\,S) \parallel (G_2 \| \#n_2\,T) \;= \\
\quad \|_{i\in I}\ ((g_i \,\&\, \neg h)\,(Q_i \parallel (G_2 \| \#n_2\,T)))\ \| \\
\quad \|_{j\in J}\ ((h_j \,\&\, \neg g)\,((G_1 \| \#n_1\,S) \parallel R_j))\ \| \\
\quad \|_{i\in I, j\in J}\ ((g_i \,\&\, h_j)\,(Q_i \parallel R_j))\ \| \\
\quad \| \#n_2\,((\#(n_1 - n_2)\,S) \parallel T)
\end{array}$$

(3). $$\begin{array}{l}
(G_1 \| \#n\,S) \parallel (G_2 \| \#n\,T) \;= \\
\quad \|_{i\in I}\ ((g_i \,\&\, \neg h)\,(Q_i \parallel (G_2 \| \#n\,T)))\ \| \\
\quad \|_{j\in J}\ ((h_j \,\&\, \neg g)\,((G_1 \| \#n\,S) \parallel R_j))\ \| \\
\quad \|_{i\in I, j\in J}\ ((g_i \,\&\, h_j)\,(Q_i \parallel R_j))\ \| \\
\quad \| \#n\,(S \parallel T)
\end{array}$$

The guarded choice is idempotent, symmetric and associative.

(*guard-1*) $G_1 \| G_1 \;=\; G_1$

(*guard-2*) $G_1 \| G_2 \;=\; G_2 \| G_1$

(*guard-3*) $(g_1\,Q_1) \| ((g_2\,Q_2) \| (g_3\,Q_3)) \;=\; ((g_1\,Q_1) \| (g_2\,Q_2)) \| (g_3\,Q_3)$

(*guard-4*) $var\, v \bullet ((\eta_v\,P) \| G_1) \;=\; var\, v \bullet G_1$

The construct *always* S executes S forever.

(*always-1*) $always\, S \;=\; S; always\, S$

Take note that *skip* is not a left zero of sequential composition in general cases, because it might filter some signal. Hereby, the following in-equation is obvious.

$$\uparrow v \;\neq\; skip; \uparrow v$$

The following definition will capture those cases where *skip* is a left zero of sequential composition.

Definition 3 (Event control insensitive)
A process P is event control insensitive if
$skip; P \;=\; P$. □

Proposition 1 *The following processes are event control insensitive.*

- $x := e$, *skip*, $\bot$, *or* $\#(t)$;
- $@(x := e)$, $\rightarrow \eta_v$;
- $P \lhd b \rhd Q$, $b * Q$, $case\,(e)\,(pt_1\,S_1) \ldots (pt_n\,S_n)$;
- $\|_{i\in I}\,(g_i\,Q_i)$, $v := g\,e$, *where no guards are event controls;*
- $P_1; P_2$, *where* P_1 *is event control insensitive;*
- $P_1 \sqcap P_2$, $P_1 \parallel P_2$, *where both* P_1 *and* P_2 *are event control insensitive;*
- *always* S, *where* S *is event control insensitive;*
- $var\, v_1, \ldots, v_n \bullet (S_1 \parallel \ldots \parallel S_n)$, *where each* S_i *is either event control insensitive, or only guarded by events with respect to variables in* $\{v_1, \ldots, v_n\}$. □

From those basic algebraic laws mentioned above, we investigate the following lemma, which will be very useful in later discussions.

Lemma 1 *Let*

$P \;=\; (\eta_u\,P_2)$, $Q \;=\; (\rightarrow \eta_u; \eta_v\,Q_2)$,

suppose sequential programs P_1, P_2, Q_1 *are event control insensitive, and variables* u, v *do not occur in* P_1 *or* Q_1, *then*

(1). $var\, u, v \bullet (P \parallel Q) \;=\; var\, u, v \bullet (P_2 \parallel (\eta_v\,Q_2))$

(2). $var\, u, v \bullet (P \parallel (Q_1; Q)) \;=\; var\, u, v \bullet (Q_1; (P \parallel Q))$

(3). $var\, u, v \bullet ((P_1; P) \parallel (Q_1; Q)) \;=\; var\, u, v \bullet ((P_1 \parallel Q_1); (P \parallel Q))$

□

Proof The proof is given in the Appendix.

We introduce an ordering relation between programs before further investigation.

Definition 4 (Refinement)
Let P, Q be Verilog processes employing the same set of variables, we say Q is a refinement of P, denoted as $P \sqsubseteq Q$, if $P \sqcap Q \;=\; P$ is algebraically provable. □

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 $\tau \in T$, $src(\tau)$ and $tgt(\tau)$ respectively represent its source and target state. Given an And-chart $c = |[s : \{p_1, \cdots, p_n\}]|$, 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 $q_{i_k} = src(\tau_{i_k})$, 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 $r_{i_k} = tgt(\tau_{i_k})$, 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 a 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,
$$resc(\tau, c) =_{df} \begin{cases} c_{[l \mapsto \mathbf{a2d}(tgt(\tau))]}, & \text{if } \tau \in T \wedge src(\tau) = p_l, \\ c_{[l \mapsto resc(\tau, p_l)]}, & \text{if } \tau \in T^*(p_l), \\ c, & \text{otherwise.} \end{cases}$$
- for an And-chart $c = |[s : \{p_1, \cdots, p_n\}]|$, and a transition τ,
$$resc(\tau, c) =_{df} \begin{cases} c_\tau, & \text{if } \tau = \&_{1 \le k \le m} \tau_{i_k} \in T(c), \\ c, & \text{otherwise.} \end{cases}$$
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 $1 \le i \le n$, $q_{i_k} = resc(\tau_{i_k}, p_{i_k})$, for $1 \le k \le m$, and $q_{i_k} = p_{i_k}$, for $m < k \le n$.

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

Definition 5 (Mapping function L) The function

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

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

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

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.
$$L(c) =_{df} \begin{cases} sink, & \text{if } T^*(c) = \emptyset \\ P, & \text{otherwise} \end{cases}$$
where
$$P =_{df} \|_{0 \le k < or\text{-}depth(c)} \|\{b_{\tau_k} \& g^i_{\tau_k} \& (\&_{1 \le j \le k} h_j) \& g^o_{\tau_k} \;\; L(resc(\tau_k, c)) \mid \tau_k \in T(active^k(c)) \wedge src(\tau_k) = active^{k+1}(c) \wedge h_j = \&\{\neg g^i_\tau \mid \tau \in T(active^{j-1}(c)) \wedge src(\tau) = active^j(c)\}\}$$

[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.

and
$active^0(c) =_{df} c$, $active^1(c) =_{df} active(c)$
$active^{i+1}(c) =_{df} active(active^i(c))$

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 $trig^+(\tau_k)$, whereas $\underline{e}_2$ are events out of $\overline{trig^-(\tau_k)}$.
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. □

Example 2 The statechart in Fig. 3 is more complicated than those in Fig. 2. It is described by:
$p = |[s : [p_1, p_{10}], p_1, \{t_1\}]|$
$p_1 = |[s_1 : [p_2, p_9], p_2, \{t_2, t_3\}]|$
$p_2 = |[s_2 : \{p_3, p_4\}]|$
$p_3 = |[s_3 : [p_5, p_6], p_5, \{t_4\}]|$
$p_4 = |[s_4 : [p_7, p_8], p_7, \{t_5\}]|$
$p_{10} = |[s_{10} : [p_{11}, p_{12}], p_{11}, \{t_6, t_7\}]|$

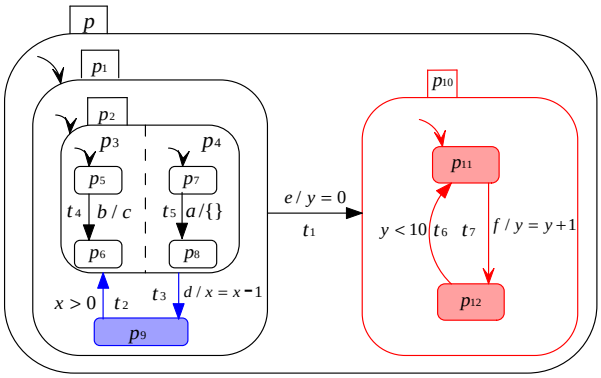

Fig. 3 A more complicated statechart

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

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

$$\mu X \bullet \left(\begin{array}{l} Q \,[\!]\, P \\ [\!](b\&\neg a\&\neg d\&\neg e\& \rightarrow c)\ (Q \,[\!]\, P \,[\!]\, (a\&\neg d\&\neg e)\ (Q [\!] P)) \\ [\!](a\&\neg b\&\neg d\&\neg e)\ (Q \,[\!]\, P \,[\!]\, (b\&\neg d\&\neg e\& \rightarrow c)\ (Q [\!] P)) \\ [\!](b\&a\&\neg d\&\neg e\& \rightarrow c)\ (Q [\!] P) \end{array} \right)$$

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.

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

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

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

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

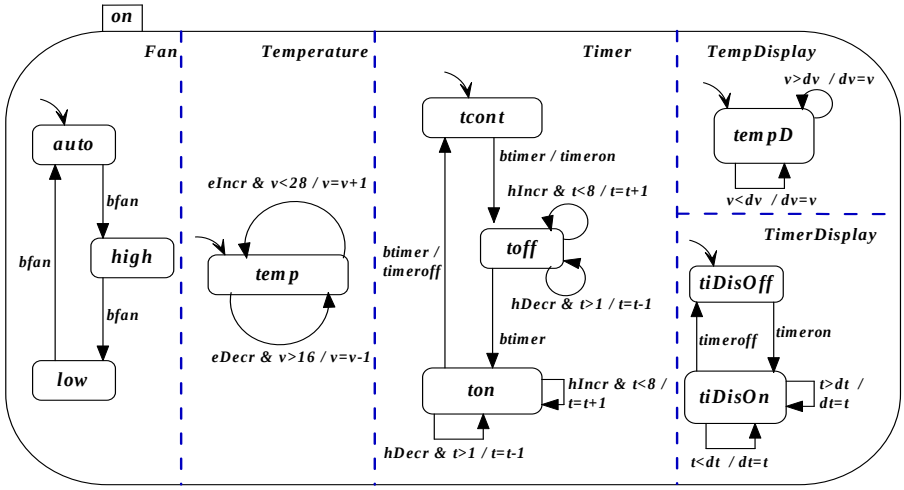

Fig. 4 An Air-Conditioner Remote Controller: the *on* state

where

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

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

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

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

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 transition in Statecharts from C to C', such that $L(C') = P'$, as illustrated in Fig. 5.*

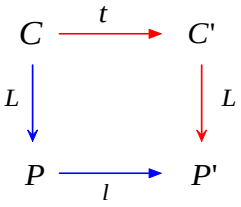

Fig. 5 Mapping function L

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 5 we know $L(C) = sink$, 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 $\tau \in T(active^k(C))$ for some $k \geq 0$ in case that all transitions of outer levels (if any) are not available, which changes the active substate of $active^k(C)$ from its source state to its target state and results in $resc(\tau, C)$; (2) otherwise, it can take a clock tick and remain its state. From Definition 5 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 $P_\tau = L(resc(\tau, C))$.
   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 5, 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 oth-

ers 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

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

It exactly accords with $L(resc(\tau, C))$. 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 5 transforms any specification in Statecharts 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 execution 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. □

5 Hardware/Software Partitioning

5.1 Partitioning Strategy

This section introduces our hardware/software partitioning strategy, which can be described in four steps, see Fig. 6.

- Before conducting the partitioning process, the programmer either (1) codes directly the kernel specification for the system in our source language (a sequential subset of Verilog which will be explained in detail next section), or (2) designs the kernel specification using Statecharts and passes it to the mapping function to generate the Verilog description.
- Then, assisted by program analysis techniques (discussed in [24]), the programmer carries out the hardware/software allocation task, i.e., marks out those parts that should be implemented by hardware and divides the variables employed by the kernel specification into two disjoint sets.
- Our hardware/software partitioning algorithm will take such a marked program as input, and deliver as output the corresponding hardware and software kernel specifications. In this step, we design and prove a collection of syntax-based splitting rules, which ensure the correctness of the partitioning process and make computer automatic partitioning possible.
- Finally, hardware/software partitioning results for the whole environment-driven system are derived from the results in the third step.

We have proposed an algebraic approach to hardware/software partitioning, which ensures the correctness of the hardware/software partitioning process and facilitates the automatic partitioning.

In what follows, we will first investigate our partitioning framework and then explore the algebraic partitioning rules.

5.2 Hardware/Software Partitioning Framework

In this section, we introduce our hardware/software partitioning framework. We depict source language and investigate the underlying target hardware-software architectures.

5.2.1 The Source Language

The source language we adopt is a sequential subset of Verilog. For clarity, we list the syntax as below.

$S ::= AC$ (primitive command)
$\mid S; S$ (sequential composition)
$\mid S \lhd b \rhd S$ (conditional)
$\mid S \sqcap S$ (non-deterministic choice)
$\mid b * S$ (iteration)
$\mid (g\, S) \,[\!]\, (g\, S)$ (guarded choice)

where

$AC ::= PC$(defined in section 3)
$\mid (v := e)_n$ (timing assignment)
$\mid \langle S \rangle$ (specific block)

The assignment statement with time constraint $(v := e)_n$ does not appear explicitly in Verilog's syntax introduced in section 3, but it is in fact a well-formed Verilog program since
$(v := e)_n \;=\; \sqcap_{0 \leq k \leq n} (v := \#k\, e)$
Moreover, the block notation in $\langle S \rangle$ has no semantical meanings.

Based on the customer's requirements, the programmer can work out the Statecharts specification and pass it to the mapping function. A Verilog specification in the above source language is then generated which will be taken as the input for the partitioning process. Alternatively, the programmer can also describe directly the kernel specification for the system to be designed in the above source language, if he/she is more familiar

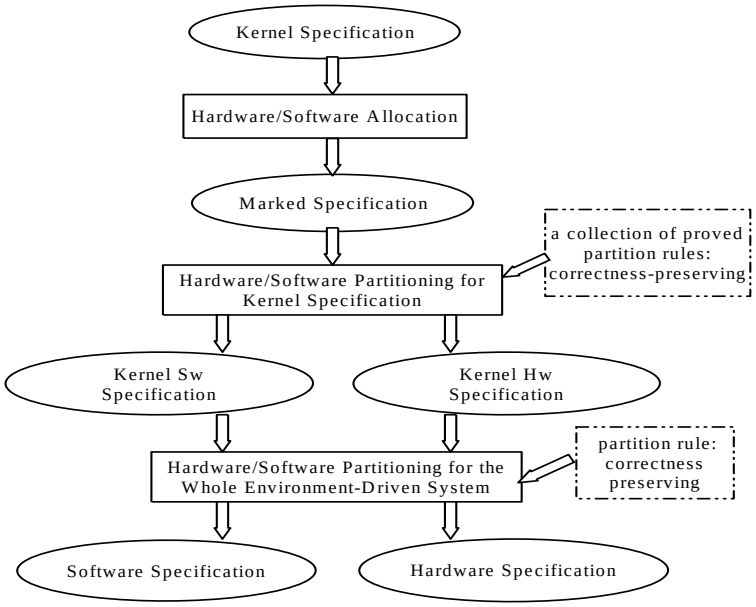

Fig. 6 Hardware/Software Partitioning Strategy

with Verilog than Statecharts. After appropriate hardware/software marking and allocation, a marked source program is then passed to the partitioning process.

5.2.2 The Underlying Target Architecture

The underlying target hardware and software components from the kernel specification will own specially-chosen forms. We adopt an event-trigger mechanism to synchronize behaviors between hardware and software, and use a shared-variable mechanism to cope with interactions between hardware and software.

The kernel part of the software specification is a member of $CP(r,a)$, a subset of Verilog programs, which is constructed by the following inductive rules.

(1). An event control insensitive process not containing variables r, a;
(2). $\rightarrow \eta_r; C; \eta_a$, where C is a member of $CP(r,a)$ not mentioning r, a;
(3). $C_1; C_2$, or $C_1 \lhd b \rhd C_2$, or $C_1 \sqcap C_2$, or $(g_1\ C_1) \,[\!]\, (g_2\ C_2)$, where $C_1, C_2, g_1, g_2 \in CP(r,a)$;
(4). $b * C$, where $C \in CP(r,a)$.

We introduce another set $CP_\varepsilon(r,a)$ comprising those processes in $CP(r,a)$ not mentioning variable ε.

As mentioned in last section, our splitting task is divided to two steps. Firstly, we design a collection of algebraic rules to refine any source program S (the kernel specification for the system) to its hardware/software decomposition

$$C_0 \parallel D_0$$

where the software component C_0 is of the form $(C; \rightarrow \eta_\varepsilon)$, C is a member of $CP_\varepsilon(r,a)$, the special event $\rightarrow \eta_\varepsilon$ is adopted for the purpose of synchronization between hardware and software, and the hardware component D_0 is subject to the following equation:

$$D_0 \;=\; \mu X \bullet ((\eta_r\ M; \rightarrow \eta_a; D_0) \,[\!]\, (\eta_\varepsilon\ \mathit{skip}))$$

where

$$M \;=_{df}\; \mathit{case}\,(\mathit{id})\,(p_1\ M_1)\ldots(p_n\ M_n)$$

is a case construct not containing r, a, ε.

We denote as $DP_\varepsilon(r,a)$ the set of processes with the same form as D_0.

To avoid any possible loss of signals at the moment when the fixed point construct (equation) is expanded, we naturally claim that an abstract event only takes place at the moment when there's no other active events at all.

Secondly, given the kernel specification S of a system, rather than considering its hardware/software partition, we deal with the decomposition for the whole system's specification

$$\Psi_f^s(S) \;=_{df}\; \mathit{always}\,(\eta_s\ S; \rightarrow \eta_f)$$

which is driven by the environmental process:

$$\mathit{Env} \;=_{df}\; \mathit{always}\,(\rightarrow \eta_s; \eta_f)$$

and derive the partitioning of $\Psi_f^s(S)$ under the environment Env as

$$\Psi_f^s(C) \parallel_{Env} D$$

where

$$P \parallel_{Env} Q \;=_{df}\; P \parallel \mathit{Env} \parallel Q$$

The software component enjoys the form

$$\Psi_f^s(C) \ =_{df} \ always\,(\eta_s\, C; \rightarrow \eta_f)$$

where C is a process from $CP(r,a)$; the hardware component D is of the form:

$$D \ =_{df} \ always\,(\eta_r\, M; \rightarrow \eta_a)$$

We denote as $DP(r,a)$ the set of processes of the same form as D.

The following theorem ensures the synchronized termination between the kernel hardware and software specifications.

Theorem 3 *We have*

$$(C_1; C_2; \rightarrow \eta_\varepsilon) \| D_0 = ((C_1; \rightarrow \eta_\varepsilon) \| D_0); ((C_2; \rightarrow \eta_\varepsilon) \| D_0)$$

for any $C_1, C_2 \in CP_\varepsilon(r,a)$ *and* $D_0 \in DP_\varepsilon(r,a)$. □

Proof By structural induction on C_1.

case 1 C_1 is event control insensitive and does not mention r or a.

(1.1) C_1 is an atomic command.

$C_1 = \perp$, the proof is trivial.

$C_1 = tg$, where tg is $@(x := e)$ or $\rightarrow \eta_x$ or $\#n$,

LHS {(*seq-6*), (*par-6*), (*guard-4*)}
$= tg\,((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {*Proposition* 1}
$= tg\,(skip;\ ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$ {(*par-6*), (*lvar-4*)}
$= tg\,((\rightarrow \eta_\varepsilon \parallel D_0);\ ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$ {(*seq-6*), (*par-6*), (*guard-4*)}
$= RHS$

(1.2) $C_1 \ = \ S_1 \sqcap S_2$

LHS {(*seq-3*)}
$= ((S_1; C_2; \rightarrow \eta_\varepsilon) \sqcap (S_2; C_2; \rightarrow \eta_\varepsilon)) \parallel D_0$ {(*par-10*)}
$= ((S_1; C_2; \rightarrow \eta_\varepsilon) \parallel D_0) \sqcap ((S_2; C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {*hypothesis*}
$= (((S_1; \rightarrow \eta_\varepsilon) \parallel D_0); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)) \sqcap (((S_2; \rightarrow \eta_\varepsilon) \parallel D_0); (C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$ {(*seq-3*)}
$= (((S_1; \rightarrow \eta_\varepsilon) \parallel D_0) \sqcap ((S_2; \rightarrow \eta_\varepsilon) \parallel D_0)); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {(*par-10*)}
$= (((S_1; \rightarrow \eta_\varepsilon) \sqcap (S_2; \rightarrow \eta_\varepsilon)) \parallel D_0); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {(*seq-3*)}
$= RHS$

(1.3) $C_1 \ = \ S_1 \lhd b \rhd S_2$.

LHS {(*seq-4*)}
$= ((S_1; C_2; \rightarrow \eta_\varepsilon) \lhd b \rhd (S_2; C_2; \rightarrow \eta_\varepsilon)) \parallel D_0$ {(*par-9*)}
$= ((S_1; C_2; \rightarrow \eta_\varepsilon) \parallel D_0) \lhd b \rhd ((S_2; C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {*hypothesis*, (*seq-4*)}
$= ((S_1; \rightarrow \eta_\varepsilon) \parallel D_0) \lhd b \rhd ((S_2; \rightarrow \eta_\varepsilon) \parallel D_0);\ ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {(*par-9*), (*seq-4*)}
$= RHS$

(1.4) $C_1 \ = \ [\!]_{i \in I}\,(g_i\, S_i)$.

LHS {(*seq-5*)}
$= ([\!]_{i \in I}\,(g_i\,(S_i; C_2; \rightarrow \eta_\varepsilon))) \parallel D_0$ {(*par-6*), (*guard-4*)}
$= [\!]_{i \in I}\,(g_i\,((S_i; C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$ {*hypothesis*}
$= [\!]_{i \in I}\,(g_i\,(((S_i; \rightarrow \eta_\varepsilon) \parallel D_0); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)))$ {(*seq-5*)}
$= [\!]_{i \in I}\,(g_i\,((S_i; \rightarrow \eta_\varepsilon) \parallel D_0));\ ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {(*par-6*), (*guard-4*), (*seq-5*)}
$= RHS$

(1.5) $C_1 \ = \ b * S$

Let $F(X) \ =_{df} \ (S; X) \lhd b \rhd skip$, and define $F^0(\perp) \ =_{df} \ \perp$, and $F^{n+1}(\perp) \ =_{df} \ F(F^n(\perp))$, for $n \geq 0$.

Then

LHS {; *is continuous*}
$= (\bigsqcup_{n \geq 0}\,(F^n(\perp); C_2; \rightarrow \eta_\varepsilon)) \parallel D_0$ {$\parallel$ *is continuous*}
$= \bigsqcup_{n \geq 0}\,((F^n(\perp); C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {*hypothesis*}
$= \bigsqcup_{n \geq 0}\,(((F^n(\perp); \rightarrow \eta_\varepsilon) \parallel D_0); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$ {$\parallel$ *and* ; *are continuous*}
$= RHS$

(1.6) $C_1 \ = \ S_1; S_2$.

(1.6.1) S_1 is a non-deterministic choice or conditional or guarded choice construct, we can respectively convert C_1 to a non-deterministic choice, or conditional or a guarded choice construct by Laws *(seq-3)*, *(seq-5)* and *(seq-4)*, which are the cases we have dealt with in (1.2), (1.3) and (1.4).

(1.6.2) S_1 is an atomic command. It's trivial when S_1 is $\perp$. We only demonstrate the proof when S_1 is a timing guard tg.

LHS {(*par-6*), (*guard-4*)}
$= tg\,((S_2; C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {*hypothesis*}
$= tg\,(((S_2; \rightarrow \eta_\varepsilon) \parallel D_0); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$ {(*seq-6*), (*par-6*), (*guard-4*)}
$= RHS$

(1.6.3) $S_1 \ = \ b * S$. Similar to (1.5).

case 2 $C_1 \ = \ \rightarrow \eta_r; C; \eta_a$.

LHS {(*par-6*), (*lvar-4*), *Proposition* 1}
$= (C; \eta_a; C_2; \rightarrow \eta_\varepsilon) \parallel (M; \rightarrow \eta_a; D_0)$ {*Lemma* 1}
$= (C \parallel M); ((\eta_a; C_2; \rightarrow \eta_\varepsilon) \parallel (\rightarrow \eta_a; D_0))$ {(*par-6*), (*lvar-4*)}
$= (C \parallel M); (skip; ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$ {(*par-6*), (*lvar-4*)}
$= (C \parallel M); (\rightarrow \eta_\varepsilon \parallel D_0); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {(*par-6*), (*lvar-4*), *Proposition* 1}
$= (C \parallel M); ((\eta_a; \rightarrow \eta_\varepsilon) \parallel (\rightarrow \eta_a; D_0)); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {*Lemma* 1}
$= ((C; \eta_a; \rightarrow \eta_\varepsilon) \parallel (M; \rightarrow \eta_a; D_0)); ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ {(*par-6*), (*lvar-4*), *Proposition* 1}
$= RHS$

case 3 C_1 is a composite construct.

(3.1) $C_1 \ = \ C^0; C^1$, similar to (1.6).
(3.2) $C_1 \ = \ C^0 \sqcap C^1$, similar to (1.2).
(3.3) $C_1 \ = \ C^0 \lhd b \rhd C^1$, analogous to (1.3).
(3.4) $C_1 \ = \ [\!]_{i \in I}\,(g_i\, C^i)$, similar to (1.4).
(3.5) $C_1 \ = \ b * C$, analogous to (1.5). □

The following corollary is directly from theorem 3.

Corollary 1 *Given* $C \in CP_\varepsilon(r,a)$ *and* $D_0 \in DP_\varepsilon(r,a)$, *we have*

$$(b * C; \rightarrow \eta_\varepsilon) \parallel D_0 \ = \ b * ((C; \rightarrow \eta_\varepsilon) \parallel D_0)$$ □

5.3 Hardware/Software Partitioning

This section specifies our hardware/software partitioning process in detail. As mentioned in section 5.1, the task is divided to two steps: hardware/software partitioning for kernel specification; decomposition of the whole system's specification. The process will be investigated in detail in the following two subsections.

5.3.1 Syntax-Based Splitting Rules for Kernel Specification

This subsection is meant to design program partitioning rules. We explore a set of splitting rules which demonstrate how to construct hardware and software parts of a program construct from those of its constituents. Meanwhile, we show how to split atomic commands.

We introduce a predicate *Split* which plays a vital role in formalizing the splitting rules.

Definition 6 (*Split*) Let $V = \{r, a, \varepsilon, id\}$. Given a program S in the source language, its hardware/software partition $((C; \rightarrow \eta_\varepsilon),\ D^0)$ is specified by the following predicate:

$$
\begin{array}{l}
Split_V(S, C, D^0) =_{df} \\
\quad (S \sqsubseteq (C; \rightarrow \eta_\varepsilon) \parallel D^0) \ \wedge \\
\quad (C \in CP_\varepsilon(r, a)) \ \wedge \ (D^0 \in DP_\varepsilon(r, a)) \ \wedge \\
\quad (V \subseteq Var(C; \rightarrow \eta_\varepsilon) \cap Var(D^0)) \ \wedge \\
\quad (V \cap OccVar(S) = \emptyset)
\end{array}
$$

where $OccVar(P)$ denotes the set of variables occurred in the program P. □

We design two set of syntax-based splitting rules in two different styles: the *bottom-up* style and the *top-down* style. The programmer can choose either of them to conduct hardware/software partitioning.

The Bottom-Up Splitting Rules The *bottom-up* approach builds the hardware component from a marked program in one step before partitioning, i.e., all services the hardware should provide are integrated at the beginning. However, it constructs the software component from those of its constituents using the following rules.

Bottom-Up Rule for Sequential Composition

$$
\frac{Split_V(S_i, C_i, D^0), i = 1, 2 \quad Var(S_1) = Var(S_2)}{Split_V(S_1; S_2,\ C_1; C_2,\ D^0)}
$$

Proof $S_1 \ ; \ S_2$ $\{; \text{ is monotonic}\}$
$\sqsubseteq$ $((C_1; \rightarrow \eta_\varepsilon) \parallel D_0);\ ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ $\{theorem\ 3\}$
$=$ $(C_1;\ C_2; \rightarrow \eta_\varepsilon) \parallel D_0$ □

Bottom-Up Rule for Conditional

$$
\frac{Split_V(S_i, C_i, D^0), i = 1, 2 \quad Var(S_1) = Var(S_2)}{Split_V(S_1 \lhd b \rhd S_2,\ C_1 \lhd b \rhd C_2,\ D^0)}
$$

Proof $S_1 \lhd b \rhd S_2$ $\{\text{conditional is mono.}\}$
$\sqsubseteq$ $((C_1; \rightarrow \eta_\varepsilon) \parallel D_0) \lhd b \rhd ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ $\{(par\text{-}9)\}$
$=$ $((C_1; \rightarrow \eta_\varepsilon) \lhd b \rhd (C_2; \rightarrow \eta_\varepsilon)) \parallel D_0$ $\{(seq\text{-}4)\}$
$=$ $((C_1 \lhd b \rhd C_2); \rightarrow \eta_\varepsilon) \parallel D_0$ □

Bottom-Up Rule for Non-Deterministic Choice

$$
\frac{Split_V(S_i, C_i, D^0), i = 1, 2 \quad Var(S_1) = Var(S_2)}{Split_V(S_1 \sqcap S_2,\ C_1 \sqcap C_2,\ D^0)}
$$

Proof $S_1 \sqcap S_2$ $\{\sqcap \text{ is mono.}\}$
$\sqsubseteq$ $((C_1; \rightarrow \eta_\varepsilon) \parallel D_0) \sqcap ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0)$ $\{(par\text{-}8)\}$
$=$ $((C_1; \rightarrow \eta_\varepsilon) \sqcap (C_2; \rightarrow \eta_\varepsilon)) \parallel D_0$ $\{(seq\text{-}3)\}$
$=$ $((C_1 \sqcap C_2); \rightarrow \eta_\varepsilon) \parallel D_0$ □

Bottom-Up Rule for Guarded Choice

$$
\frac{Split_V(S_i, C_i, D^0), i = 1, 2 \quad Var(S_1) = Var(S_2)}{Split_V((g_1\ S_1) [\!] (g_2\ S_2),\ (g_1\ C_1) [\!] (g_2\ C_2),\ D^0)}
$$

Proof $(g_1\ S_1) [\!] (g_2\ S_2)$
$\{[\!] \text{ is mono.}\}$
$\sqsubseteq$ $(g_1\ ((C_1; \rightarrow \eta_\varepsilon) \parallel D_0)) [\!] (g_2\ ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0))$
$\{(par\text{-}6),\ (guard\text{-}4)\}$
$=$ $((g_1\ (C_1; \rightarrow \eta_\varepsilon)) [\!] (g_2\ (C_2; \rightarrow \eta_\varepsilon))) \parallel D_0$
$\{(seq\text{-}5)\}$
$=$ $(((g_1\ C_1) [\!] (g_2\ C_2)); \rightarrow \eta_\varepsilon) \parallel D_0$ □

Bottom-Up Rule for Iteration

$$
\frac{Split_V(S,\ C,\ D^0)}{Split_V(b * S,\ b * C,\ D^0)}
$$

Proof $b * S$ $\{\text{loop operator is mono.}\}$
$\sqsubseteq$ $b * ((C; \rightarrow \eta_\varepsilon) \parallel D_0)$ $\{corollary\ 1\}$
$=$ $(b * C; \rightarrow \eta_\varepsilon) \parallel D_0$ □

The Top-Down Splitting Rules In the *top-down* style, both the hardware and software components of the source program are integrated from those of its constituents.

Before investigating the *top-down* splitting rules, we introduce the notion of *mergable* on hardware components from $DP_\varepsilon(r, a)$.

Definition 7 Let

$$D^i =_{df} \mu X \bullet ((\eta_r\ M^i; \rightarrow \eta_a; X) [\!] (\eta_\varepsilon\ skip))$$

where

$$M^i =_{df} case\,(id)\,(p_1^i\ M_1^i) \ldots (p_n^i\ M_{n_i}^i),\ for\ i = 1, 2$$

D^1 and D^2 are said to be *mergable*, denoted by

$$mergable(D^1, D^2)$$

if

$Var(D^1) = Var(D^2)$, and
$(p_i^1 = p_j^2)$ **implies** $M_i^1 = M_j^2$, for $1 \leq i \leq n_1$, $1 \leq j \leq n_2$.

In such a case, we define

$$D = int(D^1, D^2) \ =_{df} \ \mu X \bullet ((\eta_r \ M; \rightarrow \eta_a; X) \, [\!] \, (\eta_\varepsilon \ skip))$$

where $M \ =_{df} \ case\,(id)\,(t_1 \ M_1) \ldots (t_r \ M_r)$,
and $\{t_1, \ldots, t_r\} = \{p_1^1, \ldots, p_{n_1}^1\} \cup \{p_1^2, \ldots, p_{n_2}^2\}$,
and $\{M_1, \ldots, M_r\} = \{M_1^1, \ldots, M_{n_1}^1\} \cup \{M_1^2, \ldots, M_{n_2}^2\}$. □

First of all, we present a basic rule for hardware augmentation, from which and the *bottom-up* rules in the former section we directly obtain the corresponding *top-down* rules in all cases.

Rule for Hardware Augmentation

$$\frac{\begin{array}{c} Split_V(S, \ C, \ D) \\ mergable(D, D') \end{array}}{Split_V(S, \ C, \ int(D, D'))}$$

Proof The proof can be reached in the Appendix. □

The following top-down splitting rules are then straightforward based on the corresponding bottom-up rules and the rule for hardware augmentation above.

Top-Down Rule for Sequential Composition

$$\frac{\begin{array}{c} Split_V(S_i, C_i, D_i) \\ Var(S_1) = Var(S_2) \\ mergable(D_1, D_2) \end{array}}{Split_V(S_1; S_2, \ C_1; C_2, \ int(D_1, D_2))}$$

Top-Down Rule for Conditional

$$\frac{\begin{array}{c} Split_V(S_i, \ C_i, \ D_i) \\ Var(S_1) = Var(S_2) \\ mergable(D_1, D_2) \end{array}}{Split_V(S_1 \lhd b \rhd S_2, \ C_1 \lhd b \rhd C_2, \ int(D_1, D_2))}$$

Top-Down Rule for Non-Deterministic Choice

$$\frac{\begin{array}{c} Split_V(S_i, \ C_i, \ D_i) \\ Var(S_1) = Var(S_2) \\ mergable(D_1, D_2) \end{array}}{Split_V(S_1 \sqcap S_2, \ C_1 \sqcap C_2, \ int(D_1, D_2))}$$

Top-Down Rule for Guarded Choice

$$\frac{\begin{array}{c} Split_V(S_i, \ C_i, \ D_i) \\ Var(S_1) = Var(S_2) \\ mergable(D_1, D_2) \end{array}}{Split_V((g_1 \ S_1) \, [\!] \, (g_2 \ S_2), \ (g_1 \ C_1) \, [\!] \, (g_2 \ C_2), \ int(D_1, D_2))}$$

The top-down rule for iteration enjoys the exact form with its bottom-up rule.

Splitting Atomic Commands The details for specific blocks' partitioning are similar to discussions in [24].

For the timed assignment $(v := f(x, c))_n$, we only concentrate on the cases where both the hardware and software participate in the update of v.

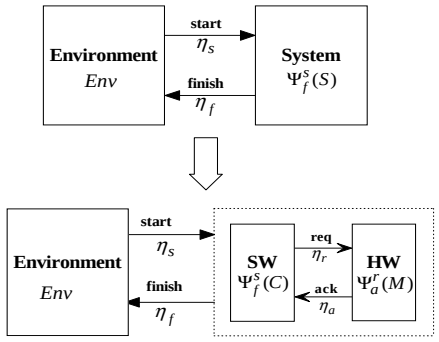

Fig. 7 Hardware/Software Partition for the Whole System

Case 1: f is a busy function, and x is allocated to hardware.

$Split_B(S = ((v := f(x, c))_n), \ C, \ D)$, where

$C \ =_{df} \ ((id := 1)_0; \rightarrow \eta_r; \eta_a; (v := ly)_0)$, and

$D \ =_{df} \ \mu X \bullet ((\eta_r \ case\,(id)\,(1\,(ly \ := \ f(x, c))_n); \rightarrow \eta_a; X) \, [\!] \, (\eta_\varepsilon \ skip))$.

Case 2: f is a busy function, but x is allocated to software.

$Split_B(S = ((v := f(x, c))_n), \ C, \ D)$, where

$C \ =_{df} \ ((id := 1)_0; (lx := x)_0; \rightarrow \eta_r; \eta_a; (v := ly)_0)$, and

$D \ =_{df} \ \mu X \bullet ((\eta_r \ case\,(id)\,(1\,(ly \ := \ f(lx, c))_n); \rightarrow \eta_a; X) \, [\!] \, (\eta_\varepsilon \ skip))$.

Case 3: f is not a busy function, but x is allocated to hardware.

$Split_B(S = ((v := f(x, c))_n), \ C, \ D)$, where

$C \ =_{df} \ ((id := 1)_0; \rightarrow \eta_r; \eta_a; (v := f(lx, c))_n)$, and

$D \ =_{df} \ \mu X \bullet ((\eta_r \ case\,(id)\,(1\,(lx := x)_0); \rightarrow \eta_a; X) \, [\!] \, (\eta_\varepsilon \ skip))$.

5.3.2 Deriving Hw/Sw Partition for an Environment-Driven System

Now we investigate hardware/software partitioning for the whole system. The partitioning process is illustrated in Fig. 7.

As discussed in sec. 5.2, suppose the whole system is specified by

$$\Psi_f^s(S) \ =_{df} \ always\,(\eta_s \ S; \rightarrow \eta_f)$$

which is driven by environment process

$$Env \ =_{df} \ always\,(\rightarrow \eta_s; \eta_f)$$

where S is the kernel specification for the system to be designed, and η_s is the start signal, η_f is the finish signal.

For a kernel specification S, suppose we have obtained its hardware/software decomposition as follows

by applying those rules in section 5.3.1:

$Split_V(S,\ C,\ D)$

where $V = \{r, a, \varepsilon, id\}$,

and $D = \mu X \bullet ((\eta_r\ M; \rightarrow \eta_a; X) \,[\!]\, (\eta_\varepsilon\ skip))$.

We design the following rule to generate the result for the partition of the whole system.

System Partitioning Rule

$$\frac{Split_V(S,\ C,\ D)}{Part(\Psi^s_f(S),\ \Psi^s_f(C),\ \Psi^r_a(M))}$$

where $Part(S, C, D) =_{df} ((S \parallel Env) \sqsubseteq (C \parallel D \parallel Env))$
$\Psi^v_u(P) =_{df} always\,(\eta_v\ P; \rightarrow \eta_u)$
$Env =_{df} always\,(\rightarrow \eta_s; \eta_f)$

Proof We define $\{always_{\,n}(S)\}$ as follows, for all $n \geq 0$:

$always_{\,0}(S) =_{df} \bot,\ always_{\,n+1}(S) =_{df} S; always_{\,n}(S)$

then by law (*always-1*), we have

$always\ S = \bigsqcup_{n\geq 0} always_{\,n}(S)$

By continuity of the parallel operator and law (*seq-2*), we only need to prove, for all $n \geq 0$,

$$(\Psi^s_f(S)_n \parallel Env_n) \sqsubseteq ((\Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel D \parallel Env_n)$$

where $\Psi^s_f(P)_n =_{df} always_{\,n}(\eta_s\ P; \rightarrow \eta_f)$
$Env_n =_{df} always_{\,n}(\rightarrow \eta_s; \eta_f)$

By mathematical induction on n.

(1). Basic step ($n = 0$).

$\Psi^s_f(S)_0 \parallel Env_0$ $\{(seq\text{-}2)\}$
$\sqsubseteq (\bot; \rightarrow \eta_\varepsilon) \parallel \bot$ $\{(par\text{-}3),\ (par\text{-}1)\}$
$= (\Psi^s_f(C)_0; \rightarrow \eta_\varepsilon) \parallel D \parallel Env_0$

(2). Inductive step ($n \rightarrow n+1$).
We first prove, for all $n \geq 0$,

$$(\Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel Env_n = always_{\,n}(C); \rightarrow \eta_\varepsilon \qquad (\dagger)$$

By an induction on n.
$n = 0$. It's straightforward by law (*par-3*) and (*seq-2*).
$n \rightarrow n+1$.

$(\Psi^s_f(C)_{n+1}; \rightarrow \eta_\varepsilon) \parallel Env_{n+1}$
$\{(par\text{-}6),\ (lvar\text{-}4),\ Proposition\ 1\}$
$= (C; \rightarrow \eta_f; \Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel (\eta_f; Env_n)$
$\{Lemma\ 1\}$
$= C; ((\rightarrow \eta_f; \Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel (\eta_f; Env_n))$
$\{(par\text{-}6),\ (lvar\text{-}4),\ Proposition\ 1\}$
$= C; ((\Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel Env_n)$
$\{hypothesis,\ (seq\text{-}1)\}$
$= always_{\,n+1}(C); \rightarrow \eta_\varepsilon$

Then, we have

$\Psi^s_f(S)_{n+1} \parallel Env_{n+1}$
$\{(par\text{-}6),\ (lvar\text{-}4),\ Proposition\ 1\}$
$= (S; \rightarrow \eta_f; \Psi^s_f(S)_n) \parallel (\eta_f; Env_n)$
$\{Lemma\ 1\}$
$= S; ((\rightarrow \eta_f; \Psi^s_f(S)_n) \parallel (\eta_f; Env_n))$
$\{(par\text{-}6),\ (lvar\text{-}4),\ Proposition\ 1\}$
$= S; (\Psi^s_f(S)_n \parallel Env_n)$
$\{precondition,\ ;\ is\ mono.\}$
$\sqsubseteq ((C; \rightarrow \eta_\varepsilon) \parallel D); (\Psi^s_f(S)_n \parallel Env_n)$
$\{hypothesis\}$
$\sqsubseteq ((C; \rightarrow \eta_\varepsilon) \parallel D); ((\Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel D \parallel Env_n)$
$\{(\dagger)\}$
$= ((C; \rightarrow \eta_\varepsilon) \parallel D); ((always_{\,n}(C); \rightarrow \eta_\varepsilon) \parallel D)$
$\{Theorem\ 3\}$
$= (always_{\,n+1}(C); \rightarrow \eta_\varepsilon) \parallel D$
$\{(\dagger)\}$
$= (\Psi^s_f(C)_{n+1}; \rightarrow \eta_\varepsilon) \parallel D \parallel Env_{n+1}$

□

6 Related Work

Statecharts semantics 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 [7; 15; 18; 19; 22; 30]. Many of these works adopt the simpler synchronous model. The work in [7] 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.

Verilog semantics 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 [12; 32; 33; 8] are all based on Gordon's interpretation on simulation cycles [3]. A simple operational semantics is given in [12]. Zhu, Bowen and He [32; 33] investigate the consistency between Verilog's operational and denotational semantics, while He [8] explores a program algebra for Verilog and its connection with both operational and denotational semantics. Most recently Zhu [31] provides a more complete investigation on unifying different semantic models for Verilog-like languages.

Linking Statecharts with other formalisms Some of related works on connecting Statecharts with other formalisms are presented in [1; 2; 20; 27; 29; 26]. Beauvais et.al. [1] and Seshia et.al. [27] 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 [7] lacking correctness proofs. The authors of [2; 20] transform variants of Statecharts into hierarchical timed automata and use tools (UPPAAL, SPIN) to model check Statecharts properties. Also, [29] based on the denotational semantics [15] 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 [26]. 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.

Algebraic approach to Hardware/software partitioning The algebraic approach advocated in this paper to verify the correctness of the partitioning process has been successfully employed in the **ProCoS** project. The original **ProCoS** project [11] concentrated almost exclusively on the verification of standard compiler of a high-level programming language based on Occam down to a microprocessor based on Transputer [10]. Sampaio showed how to reduce the compiler design task to program transformation [25]. Towards the end of the first phase of the project, Ian Page *et al* made rapid advance in the development of hardware compilation technique using an Occam-like language targeted towards FPGAs [21], and He Jifeng *et al* provided a formal verification of the hardware compilation scheme within the algebra of Occam programs [9].

Some works have suggested the use of formal methods for the partitioning process [28; 24]. In [28], Silva *et al* provide a formal strategy for carrying out the splitting phase automatically, and present an algebraic proof for its correctness. However, the splitting phase delivers a large number of simple processes, and leaves the hard task of clustering these processes into hardware and software components to the clustering phase and the joining phase. Furthermore, additional channels and local variables introduced in the splitting phase increase the data flow between hardware and software components. In our former work [24], an algebraic approach is proposed to partition a specification into hardware and software in one step and as well verify the correctness of the partition process. However, that approach is based on algebraic laws of the high level communicating language Occam, which leaves rather a long distance to go through in hardware/software co-synthesis phase. In this paper, the distance has been shortened by adopting Verilog as the language.

7 Conclusion

In this paper we present a formal approach to hardware/software co-specification, starting from the Statecharts visual formalism. We have made the following main contributions.

- Compositional operational semantics for Statecharts. We have explored 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.
- A semantics-preserving linking between Statecharts and Verilog. We have defined a syntax-directed function to map Statecharts to Verilog programs. Based on the operational semantics for Statecharts and Verilog, we have proved the linking function is a semantics-preserving homomorphism.
- An algebraic approach to hardware/software partitioning in Verilog. We have worked out a collection of formal rules to split a specification (in Verilog) into hardware and software sub-specifications. The partitioning process has been proved sound using Verilog algebra.

  We adopt a sequential imperative subset of Verilog as our source language, and allow it to contain time constraints, so as to describe timing specification. We confine target hardware and software specifications in specially chosen subsets of Verilog, and use Verilog's event-trigger mechanism to synchronize behaviors between them. Whereas, communications between hardware and software are based on Verilog's shared variable mechanism, which will facilitate the subsequent hardware/software co-synthesis, and make it possible to adopt bus techniques to implement interactions between hardware and software.

  Moreover, this paper not only develops a collection of splitting rules to partition a source program into hardware and software components, but also discuss hardware/software partitioning for the whole system which takes the source program as its kernel specification. The system is specified by Verilog's *always* constructs and its execution is driven by an environment process. Such systems widely exist in our daily life, *embedded systems* are of this kind. Developing a partitioning rule for such systems will be very helpful for us to investigate correctness-preserved design of embedded systems.

Acknowledgements The first part of the work was done when the first author was affiliated with Singapore-MIT Alliance, National University of Singapore. The second part of the work was done when the first author was doing his Ph.D in School of Mathematical Sciences, Peking University.

References

1. Beauvais J.-R. et. al., A Translation of Statecharts to Signal/DC+, Technical Report, IRISA, 1997.
2. David A., Möller O., and Wang Y., 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. Gordon M, 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. Gordon M, Relating Event and Trace Semantics of Hardware Description Languages, *The Computer Journal*, pp. 27–36, Vol. 45, No. 1, 2002.
5. Harel D., Statecharts: a Visual Formalism for Complex Systems, *Science of Computer Programming*, vol.8, no.3, pp. 231–274, 1987.
6. Harel D., On Visual Formalisms, *Communications of the ACM*, Vol. 31, No. 5, pp. 541–530, 1988.
7. Harel D. and Naamad A., The STATEMATE Semantics of Statecharts, *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 4, pp. 293–333, October, 1996.
8. He J., 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.
9. He J., Page I., and Bowen J., A Provable Hardware Implementation of Occam, *LNCS* 711, pp. 693–703, 1993.
10. He J. and Bowen J., Specification, Verification and Prototyping of an Optimised Compiler, *Formal Aspect of Computing* 6, pp. 643–658, 1994.
11. He J. *et al*, Provably Correct Systems, *LNCS* 863, pp. 288–335, 1994.
12. He J. and Xu Q., 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.
13. He J. and Zhu H., Formalising Verilog, in *the proc. of ICECS 2000*, IEEE Computer Society Press, pp. 412–415, Lebanon, Dec. 2000.
14. Hoare C.A.R. and He J., *Unifying Theories of Programming*, Prentice Hall, 1998.
15. Hooman J.J.M., Ramesh S., and de Roever W.P., A Compositional Axiomatization of Statecharts, *Theoretical Computer Science* 101, pp. 289–335, 1992.
16. IEEE Computer Society, *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE std 1364-1995)*, 1995.
17. Iyoda J. and He J., Towards an Algebraic Synthesis of Verilog, in *the proc of ERSA'2001*, Las Vegas, USA, 2001.
18. Lüttgen G., von der Beeck M., and Cleaveland R., A Compositional Approach to Statecharts Semantics, NASA/CR-2000-210086, ICASE Report No.2000-12, March, 2000.
19. Maggiolo-Schettini A., Peron A., and Tini S., Equivalences of Statecharts, in *7th International Conference on Concurrency Theory (CONCUR'96)*, Pisa, Italy, Aug. 1996, LNCS 1119, pp.687–702, Springer-Verlag.
20. Mikk E., Lakhnech Y., Siegel M., and Holzmann G., Implementing Statecharts in Promela/SPIN, in the *Proc. of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, IEEE Computer Society, 1999.
21. Page I. and Luk W., Compiling Occam into FPGAs, in *FPGAs*, eds., W. Moore and W. Luk, pp. 271–283, Abingdon EE&CS books, 1991.
22. Pnueli A. and Shalev M., 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.
23. Qin S. and He J., 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.
24. Qin S. and He J., Partitioning Program into Hardware and Software, in *the Proc of APSEC 2001*, IEEE Computer Society Press, pp. 309–316, Macau, 4–7 December, 2001.
25. Sampaio A., *An Algebraic Approach to Compiler Design*, World Scientific, 1997.
26. Sekerinski E. and Zurob R., 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.
27. Seshia S., Shyamasundar R., Bhattacharjee A., and Dhodapkar S., 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.
28. Silva L., Sampaio A., and Barros E., A Normal Form Reduction Strategy for Hardware/software Partitioning, *Formal Methods Europe (FME) 97, LNCS* 1313, pp. 624–643, 1997.
29. Sowmya A. and Ramesh S., Extending Statecharts with Temporal Logic, *IEEE Transactions on Software Engineering*, Vol. 24, No. 3, March, 1998.
30. von der Beeck M., 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.
31. Zhu H., Linking the Semantics of a Multithreaded Discrete Event Simulation Language, Ph.D thesis, London South Bank University, UK. February 2005.
32. Zhu H., Bowen J., and He J., From Operational Semantics to Denotational Semantics for Verilog, in *the Proc. of CHARME 2001, LNCS* 2144, pp. 449–464.
33. Zhu H., Bowen J., and He J., 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.
34. Zhu H., Bowen J., and He J., Deriving Operational Semantics from Denotational Semantics for Verilog, in *the proc. of APSEC 2001*, IEEE Computer Society Press, pp. 177–184, Macau, 4–7 December, 2001.
35. Zhu H. and He J., A DC-based Semantics for Verilog, in *the proc. of the International Conference on Software: Theory and Practice (ICS2000)*, pp. 421–432, Beijing, 21–24 August, 2000.

Appendix

Proof of Lemma 1

Proof (1). LHS $\{(par\text{-}6)\}$
$= \; var\, u, v \bullet ((\eta_u (P_2 \parallel Q)) \,[\!]\, (\rightarrow \eta_u; (P \parallel (\eta_v; Q_2))))$ $\{(guard\text{-}4)\}$
$= \; var\, u, v \bullet (\rightarrow \eta_u; (P \parallel (\eta_v; Q_2)))$ $\{(par\text{-}6)\}$
$= \; var\, u, v \bullet (\rightarrow \eta_u; (\eta_u; (P_2 \parallel \eta_v\, Q_2))) \,[\!]\, (\eta_v; (P \parallel Q_2)))$ $\{(lvar\text{-}4)\}$
$= \; var\, u, v \bullet (skip; (P_2 \parallel \eta_v\, Q_2))$ $\{Proposition\ 1\}$
$= \; RHS$

(2). By structural induction on Q_1.

(2.1) Q_1 is an atomic command.

It is obvious when $Q_1 = stop$ or $Q_1 = \perp$.

$Q_1 = tg$, where tg is one of the following forms $@(x := e)$, $\rightarrow \eta_x$, $\#n$. We have

LHS $\{(par\text{-}6)\}$
$= \ var\ u, v \bullet ((\eta_u; (P_2 \parallel (Q_1; Q))) [\!] (tg; (P \parallel Q)))$ $\{(guard\text{-}4)\}$
$= \ RHS$

(2.2) $Q_1 = S_1 \sqcap S_2$.

LHS $\{(seq\text{-}3)\}$
$= \ var\ u, v \bullet (P \parallel ((S_1; Q) \sqcap (S_2; Q)))$ $\{(par\text{-}1),\ (par\text{-}10)\}$
$= \ var\ u, v \bullet ((P \parallel (S_1; Q)) \sqcap (P \parallel (S_2; Q)))$ $\{hypothesis\}$
$= \ var\ u, v \bullet ((S_1; (P \parallel Q)) \sqcap (S_2; (P \parallel Q)))$ $\{(seq\text{-}3)\}$
$= \ RHS$

(2.3) $Q_1 = S_1 \lhd b \rhd S_2$.

LHS $\{(seq\text{-}4)\}$
$= \ var\ u, v \bullet (P \parallel ((S_1; Q) \lhd b \rhd (S_2; Q)))$ $\{(par\text{-}9)\}$
$= \ var\ u, v \bullet ((P \parallel (S_1; Q)) \lhd b \rhd (P \parallel (S_2; Q)))$ $\{hypothesis\}$
$= \ var\ u, v \bullet ((S_1; (P \parallel Q)) \lhd b \rhd (S_2; (P \parallel Q)))$ $\{(seq\text{-}4)\}$
$= \ RHS$

(2.4) $Q_1 \ = \ [\!]_{i \in I} (g_i\ S_i)$.

LHS $\{(seq\text{-}5)\}$
$= \ var\ u, v \bullet (P \parallel ([\!]_{i \in I} (g_i\ (S_i; Q))))$ $\{(par\text{-}6),\ (guard\text{-}4)\}$
$= \ var\ u, v \bullet ([\!]_{i \in I} (g_i\ (P \parallel (S_i; Q))))$ $\{hypothesis\}$
$= \ var\ u, v \bullet ([\!]_{i \in I} (g_i\ (S_i; (P \parallel Q))))$ $\{(seq\text{-}5)\}$
$= \ RHS$

(2.5) $Q_1 \ = \ b * S$.

Let

$F(X) \ =_{df} \ (S; X) \lhd b \rhd skip,$

For $n \geq 0$ we define

$F^0(\bot) \ =_{df} \ \bot,$

$F^{n+1}(\bot) \ =_{df} \ F(F^n(\bot)).$

Then

LHS $\{continuity\ of\ ;\}$
$= \ var\ u, v \bullet (P \parallel (\bigsqcup_{n \geq 0} (F^n(\bot); Q)))$ $\{continuity\ of\ \parallel\}$
$= \ var\ u, v \bullet (\bigsqcup_{n \geq 0} (P \parallel (F^n(\bot); Q)))$ $\{hypothesis\}$
$= \ var\ u, v \bullet (\bigsqcup_{n \geq 0} (F^n(\bot); (P \parallel Q)))$ $\{continuity\ of\ \parallel\ and\ ;\}$
$= \ RHS$

(2.6) $Q_1 \ = \ S_1; S_2$.

(2.6.1) S_1 is one of the following forms: $S_{10} \sqcap S_{11}$, $S_{10} \lhd b \rhd S_{11}$, $(g_1\ S_{10}) [\!] (g_2\ S_{11})$. By laws $(seq\text{-}3)$ $(seq\text{-}5)$, and $(seq\text{-}4)$ we can convert Q_1 to non-deterministic choice, conditional, and guarded-choice, respectively. It then follows from (2.2),(2.3), and (2.4).

(2.6.2) S_1 is an atomic command. It is straightforward when S_1 is $\bot$ or $stop$. Let us consider $S_1 = tg$.

LHS $\{(par\text{-}6),\ (guard\text{-}4)\}$
$= \ var\ u, v \bullet (tg; (P \parallel (S_2; Q)))$ $\{hypothesis\}$
$= \ RHS$

(2.6.3) $S_1 \ = \ b * S$.

Similar to (2.5), we have

LHS $\{continuity\ of\ ;\}$
$= \ var\ u, v \bullet (P \parallel (\bigsqcup_{n \geq 0} (F^n(\bot); S_2; Q)))$ $\{continuity\ of\ \parallel\}$
$= \ var\ u, v \bullet (\bigsqcup_{n \geq 0} (P \parallel (F^n(\bot); S_2; Q)))$ $\{hypothesis\}$
$= \ var\ u, v \bullet (\bigsqcup_{n \geq 0} (F^n(\bot); S_2; (P \parallel Q)))$ $\{continuity\ of\ \parallel\ and\ ;\}$
$= \ RHS$

(3). By structural induction on P_1.

(3.1) P_1 is an atomic command. The proof can be done by a structural induction on Q_1. Similar to the proof of (2).

(3.2) $P_1 = S_1 \sqcap S_2$.

LHS $\{(seq\text{-}3)\}$
$= \ var\ u, v \bullet (((S_1; P) \sqcap (S_2; P)) \parallel (Q_1; Q))$ $\{(par\text{-}10)\}$
$= \ var\ u, v \bullet (((S_1; P) \parallel (Q_1; Q)) \sqcap ((S_2; P) \parallel (Q_1; Q)))$ $\{hypothesis\}$
$= \ var\ u, v \bullet (((S_1 \parallel Q_1); (P \parallel Q)) \sqcap ((S_2 \parallel Q_1); (P \parallel Q)))$ $\{(seq\text{-}3)\}$
$= \ RHS$

(3.3) $P_1 = S_1 \lhd b \rhd S_2$.

LHS $\{(seq\text{-}4)\}$
$= \ var\ u, v \bullet (((S_1; P) \lhd b \rhd (S_2; P)) \parallel (Q_1; Q))$ $\{(par\text{-}9)\}$
$= \ var\ u, v \bullet ((((S_1; P) \parallel (Q_1; Q)) \lhd b \rhd ((S_2; P) \parallel (Q_1; Q)))$ $\{hypothesis\}$
$= \ var\ u, v \bullet ((((S_1 \| Q_1); (P \| Q)) \lhd b \rhd ((S_2 \| Q_1); (P \| Q)))$ $\{(seq\text{-}4),\ (par\text{-}9)\}$
$= \ RHS$

(3.4) $P_1 \ = \ [\!]_{i \in I} (g_i\ S_i)$.

We first have LHS $\{(seq\text{-}5)\}$
$= \ var\ u, v \bullet (([\!]_{i \in I} (g_i\ (S_i; P))) \parallel (Q_1; Q))$

Then, apply structural induction on Q_1. The proof is similar to the proof in (2).

(3.5) $P_1 \ = \ b * S$.

Let

$F(X) \ =_{df} \ (S; X) \lhd b \rhd skip,$

For $n \geq 0$ we define

$F^0(\bot) \ =_{df} \ \bot,$

$F^{n+1}(\bot) \ =_{df} \ F(F^n(\bot)).$

Then

$$
\begin{array}{ll}
& LHS \quad \{continuity\ of\ ;\} \\
= & var\ u, v \bullet ((\bigsqcup_{n\geq 0} (F^n(\bot); P)) \parallel (Q_1; Q)) \\
& \quad \{continuity\ of\ \parallel\} \\
= & var\ u, v \bullet (\bigsqcup_{n\geq 0} ((F^n(\bot); P) \parallel (Q_1; Q))) \ \{hypothesis\} \\
= & var\ u, v \bullet (\bigsqcup_{n\geq 0} ((F^n(\bot) \parallel Q_1); (P \parallel Q))) \\
& \quad \{continuity\ of\ \parallel\ and\ ;\} \\
= & RHS
\end{array}
$$

(3.6) $P_1 \ = \ S_1; S_2$.

(3.6.1) S_1 is one of the following forms: $S_{10} \sqcap S_{11}$, $S_{10} \lhd b \rhd S_{11}$, $(g_1\ S_{10}) [\!] (g_2\ S_{11})$. By laws (*seq-3*) (*seq-5*), and (*seq-4*), we can convert P_1 to non-deterministic choice, conditional, guarded-choice, respectively. The proof then follows from (3.2),(3.3), and (3.4).

(3.6.2) S_1 is an atomic command. It is obvious when S_1 is $\bot$ or *stop*. For the case $S_1 = tg$, the proof follows from a structural induction on Q_1. The proof is similar to that in (2).

(3.6.3) $S_1 \ = \ b * S$.

Similar to (3.5), we have

$$
\begin{array}{ll}
& LHS \quad \{continuity\ of\ ;\} \\
= & var\ u, v \bullet ((\bigsqcup_{n\geq 0} (F^n(\bot); S_2; P)) \parallel (Q_1; Q)) \\
& \quad \{continuity\ of\ \parallel\} \\
= & var\ u, v \bullet (\bigsqcup_{n\geq 0} ((F^n(\bot); S_2; P) \parallel (Q_1; Q))) \\
& \quad \{hypothesis\} \\
= & var\ u, v \bullet (\bigsqcup_{n\geq 0} (((F^n(\bot); S_2) \parallel Q_1); (P \parallel Q))) \\
& \quad \{continuity\ of\ \parallel\ and\ ;\} \\
= & RHS \qquad \Box
\end{array}
$$

Proof of the Rule for Hardware Augmentation

Proof For simplicity, let us denote $int(D, D')$ as $\hat{D}$. We need to prove

$$(C; \rightarrow \eta_\varepsilon) \parallel D \ = \ (C; \rightarrow \eta_\varepsilon) \parallel \hat{D}$$

By structural induction on C.

case 1 C does not contain r or a, and C is event control insensitive.

(1.1) C is an atomic command.

$C = stop$ or $C = \bot$, trivial.

$C = tg$, where tg is one of the following forms $@(x := e)$, $\rightarrow \eta_x$, $\#n$. We have

$$
\begin{array}{ll}
& LHS \quad \{(par\text{-}6),\ (guard\text{-}4)\} \\
= & tg;\ (\rightarrow \eta_\varepsilon \parallel D) \quad \{(lvar\text{-}4)\} \\
= & tg;\ skip; \quad \{(lvar\text{-}4)\} \\
= & tg;\ (\rightarrow \eta_\varepsilon \parallel \hat{D}) \quad \{(par\text{-}6),\ (guard\text{-}4)\} \\
= & RHS
\end{array}
$$

(1.2) $C \ = \ S_1 \sqcap S_2$.

$$
\begin{array}{ll}
& LHS \quad \{(seq\text{-}3)\} \\
= & ((S_1; \rightarrow \eta_\varepsilon) \sqcap (S_2; \rightarrow \eta_\varepsilon)) \parallel D \quad \{(par\text{-}10))\} \\
= & ((S_1; \rightarrow \eta_\varepsilon) \parallel D) \sqcap ((S_2; \rightarrow \eta_\varepsilon) \parallel D) \quad \{hypothesis\} \\
= & ((S_1; \rightarrow \eta_\varepsilon) \parallel \hat{D}) \sqcap ((S_2; \rightarrow \eta_\varepsilon) \parallel \hat{D}) \quad \{(par\text{-}10)\} \\
= & RHS
\end{array}
$$

(1.3) $C \ = \ S_1 \lhd b \rhd S_2$.

$$
\begin{array}{ll}
& LHS \quad \{(seq\text{-}4)\} \\
= & ((S_1; \rightarrow \eta_\varepsilon) \lhd b \rhd (S_2; \rightarrow \eta_\varepsilon)) \parallel D \quad \{(par\text{-}9)\} \\
= & ((S_1; \rightarrow \eta_\varepsilon) \parallel D) \lhd b \rhd ((S_2; \rightarrow \eta_\varepsilon) \parallel D) \quad \{hypothesis\} \\
= & ((S_1; \rightarrow \eta_\varepsilon) \parallel \hat{D}) \lhd b \rhd ((S_2; \rightarrow \eta_\varepsilon) \parallel \hat{D}) \quad \{(par\text{-}9)\} \\
= & RHS
\end{array}
$$

(1.4) $C \ = \ [\!]_{i\in I} (g_i\ S_i)$.

$$
\begin{array}{ll}
& LHS \quad \{(seq\text{-}5)\} \\
= & ([\!]_{i\in I} (g_i\ (S_i; \rightarrow \eta_\varepsilon))) \parallel D \quad \{(par\text{-}6),\ (guard\text{-}4)\} \\
= & [\!]_{i\in I} (g_i\ ((S_i; \rightarrow \eta_\varepsilon) \parallel D)) \quad \{hypothesis\} \\
= & [\!]_{i\in I} (g_i\ ((S_i; \rightarrow \eta_\varepsilon) \parallel \hat{D})) \quad \{(par\text{-}6)\} \\
= & RHS
\end{array}
$$

(1.5) $C \ = \ b * S$.

Let

$F(X) \ =_{df} \ (S; X) \lhd b \rhd skip$,

For $n \geq 0$, we define

$F^0(\bot) \ =_{df} \ \bot$,

$F^{n+1}(\bot) \ =_{df} \ F(F^n(\bot))$.

Then

$$
\begin{array}{ll}
& LHS \quad \{continuity\ of\ ;\} \\
= & (\bigsqcup_{n\geq 0} (F^n(\bot); \rightarrow \eta_\varepsilon)) \parallel D \quad \{continuity\ of\ \parallel\} \\
= & \bigsqcup_{n\geq 0} ((F^n(\bot); \rightarrow \eta_\varepsilon) \parallel D) \quad \{hypothesis\} \\
= & \bigsqcup_{n>0} ((F^n(\bot); \rightarrow \eta_\varepsilon) \parallel \hat{D}) \quad \{continuity\ of\ \parallel\} \\
= & RHS
\end{array}
$$

(1.6) $C \ = \ S_1; S_2$.

(1.6.1) S_1 is one of the following forms $S_{10} \sqcap S_{11}$, $S_{10} \lhd b \rhd S_{11}$, $(g_1\ S_{10}) [\!] (g_2\ S_{11})$. By laws (*seq-3*), (*seq-5*), and (*seq-4*) we can convert C into non-deterministic choice, conditional, and guarded-choice, respectively. The proof then follows from (1.2), (1.3), and (1.4).

(1.6.2) S_1 is an atomic command. It is trivial when S_1 is $\bot$ or *stop*. The case $S_1 = tg$ is dealt with in what follows.

$$
\begin{array}{ll}
& LHS \quad \{(par\text{-}6),\ (guard\text{-}4)\} \\
= & tg;\ (S_2 \parallel D) \quad \{hypothesis\} \\
= & tg;\ (S_2 \parallel \hat{D}) \quad \{(par\text{-}6),\ (guard\text{-}4)\} \\
= & RHS
\end{array}
$$

(1.6.3) $S_1 \ = \ b * S$.

Similar to (1.5) we have

$$
\begin{array}{ll}
& LHS \qquad \{continuity\ of\ ;\} \\
= & (\bigsqcup_{n\geq 0}(F^n(\bot); S_2; \rightarrow\eta_\varepsilon)) \parallel D \qquad \{continuity\ of\ \parallel\} \\
= & \bigsqcup_{n\geq 0}((F^n(\bot); S_2 \rightarrow\eta_\varepsilon) \parallel D) \qquad \{hypothesis\} \\
= & \bigsqcup_{n\geq 0}(((F^n(\bot); S_2; \rightarrow\eta_\varepsilon) \parallel \hat{D}) \qquad \{continuity\ of\ \parallel \\
& and\ ;\} \\
= & RHS
\end{array}
$$

case 2 $C = \rightarrow\eta_r; C_0; \eta_a$.

$$
\begin{array}{ll}
& LHS \qquad \{(lvar\text{-}4),\ event\ control\ insensitive\} \\
= & (C_0; \eta_a; \rightarrow\eta_\varepsilon) \parallel (M; \rightarrow\eta_a; D) \qquad \{Lemma\ 1\} \\
= & (C_0 \parallel M); ((\eta_a; \rightarrow\eta_\varepsilon) \parallel (\rightarrow\eta_a; D)) \\
& \qquad \{(lvar\text{-}4),\ event\ control\ insensitive\} \\
= & (C_0 \parallel M); skip; (\rightarrow\eta_\varepsilon \parallel D) \qquad \{(lvar\text{-}4)\} \\
= & (C_0 \parallel M); skip; (\rightarrow\eta_\varepsilon \parallel \hat{D}) \qquad \{(lvar\text{-}4)\} \\
= & (C_0 \parallel M); ((\eta_a; \rightarrow\eta_\varepsilon) \parallel (\rightarrow\eta_a; \hat{D})) \qquad \{Lemma\ 1\} \\
= & ((C_0; \eta_a; \rightarrow\eta_\varepsilon) \parallel (M; \rightarrow\eta_a; \hat{D})) \\
& \qquad \{(lvar\text{-}4),\ event\ control\ insensitive\} \\
= & RHS
\end{array}
$$

case 3 C is one of the following forms.

(3.1) $C = C^0; C^1$, analgous to that in (1.6).

(3.2) $C = C^0 \sqcap C^1$, analgous to that in (1.2).

(3.3) $C = C^0 \lhd b \rhd C^1$, analgous to that in (1.3).

(3.4) $C = \|_{i\in I}(g_i\, C^i)$, analgous to that in (1.4).

(3.5) $C = b * C$, analgous to that in (1.5). □