This full text version, available on TeesRep, is the post-print (final version prior to publication) of:


For details regarding the final published version please click on the following DOI link: http://dx.doi.org/10.1007/s11334-005-0020-2

When citing this source, please use the final published version as above.

This document was downloaded from http://tees.openrepository.com/tees/handle/10149/110727

Please do not use this version for citation purposes.

All items in TeesRep are protected by copyright, with all rights reserved, unless otherwise indicated.
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 circuitry. 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
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 behavioural 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. Booman 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:

\( S \): a set of names used to denote Statecharts which is large enough to prevent name conflicts.

\( \Pi_e \): 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 \( \Pi_e \), i.e. \( \overline{\Pi_e} = \{ \overline{e} \mid e \in \Pi_e \} \), where \( \overline{e} \) denotes the negated counterpart of event \( e \), and we assume \( \overline{\overline{e}} = e \).

\( \Pi_a \): the set of all assignment actions of the form \( v = \exp \).

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

\( \mathcal{T} \): the set of transitions, which is a subset of \( S \times 2^{\Pi_e \cup \Pi_a} \times 2^{\Pi_e \cup \Pi_a} \times B \times S \), where \( B \) 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.

| Basic | \( S \rightarrow SC \) |
| Basic(s) | \( \Rightarrow \{ [s] \} \) |
| Or(s, \( SC \times SC \times \mathcal{T} \rightarrow SC \) | \( \Rightarrow \{ [s] \mid [s] \in SC \} \) |
| And(s, \( SC \times SC \rightarrow SC \) | \( \Rightarrow \{ [s] \mid [s] \in SC \} \) |

Some informal explanations follow:

– Basic(s) denotes a basic statechart named \( s \).

– Or(s, \[ p_1, \ldots, p_n \], \( p, T \) ) represents an Or-statechart with a set of subcharts \( \{ p_1, \ldots, p_n \} \), where \( p_1 \) is the default substate, \( p \) is the active substate, \( T \) is composed of all possible transitions among immediate substates of \( s \).

– And(s, \[ p_1, \ldots, p_n \] ) is an And-statechart named \( s \), which contains a set of orthogonal (concurrent) substates \( \{ p_1, \ldots, p_n \} \).

2.2 Operational Transition Rules

The configuration of computation is defined by a triple \( (p, \sigma, E_{in}) \), where

– \( p \) is the syntax of the statechart of interest.
– \( \sigma \) 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. \( \left[ [s], \sigma, E \right] \xrightarrow{\tau} \left[ [s], \sigma, 0 \right] \)

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.

\[
p = \left[ \left[ s : [p_1, \ldots, p_n] ; p ; T \right] \right] \in E_0(p, E) \wedge \sigma(b)
\]

2. \( \left[ p, \sigma, E \right] \xrightarrow{\text{adv}} \left[ \left[ p_0 \right] ; p \xrightarrow{\text{adv}} (\text{tgt}(\tau) \downarrow \sigma', (E - \text{trig}(\tau)) \cup a'(\tau)) \right] \)

where

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

\( a'(\tau) \subseteq \Pi \) represents all events generated by transition \( \tau \), whereas \( a(\tau) \) denotes a single assignment action \( v = ex \) generated by \( \tau \). 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 \( \sigma \) 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.

\[
\tau \in En([s : [p_1, \ldots, p_n], p_i, T]), E) \quad \text{iff} \quad \tau \in T \land \text{src}(\tau) = p_i \land \text{trig}^+(\tau) \subseteq E \land \text{trig}^-(\tau) \cap E = \emptyset.
\]

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

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

\[
a2d([s]) = [s]
\]

\[
a2d([s : [p_1, \ldots, p_n], p_i, T]) = [s : [p_1, \ldots, p_n], a2d(p_1), T]]
\]

\[
a2d([s : [p_1, \ldots, p_n]]) = [s : [a2d(p_1), \ldots, a2d(p_n)])
\]

The substitution \( p_{[p_0 \rightarrow p_n]} \) for an Or-statechart \( p = [[s : [p_1, \ldots, p_n], p_i, T]] \) is defined by:

\[
p_{[p_0 \rightarrow p_n]} = [s : [p_1, \ldots, p_0, p_n, T]]
\]

**Discussion:** In rule 2, those events that are used to trigger \( \tau \) are consumed by \( \tau \) 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 \( \tau_1 \) 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.

\[
p = [[s : [p_1, \ldots, p_n], p_i, T]]
\]

\[
En(p, E) \neq \emptyset
\]

\[
\langle p, \sigma, E \rangle \xrightarrow{\tau \in k} \langle p_{[p \rightarrow p_0]}, \sigma', (E - \text{trig}^+(\tau)) \cup \sigma'(\tau) \rangle
\]

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

\[
En^*(p, E) \neq \emptyset
\]

\[
\langle p = [[s : [p_1, \ldots, p_n], p_i, T]], \sigma, E] \xrightarrow{\gamma} \langle p, \sigma, \emptyset \rangle
\]

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

\[
En^*([s], E) = \emptyset, \text{for any basic state} \ [s];
\]

\[
En^*([s : [p_1, \ldots, p_n], p_i, T], E) = \emptyset \cup En^*(p_i, E); \quad En^*([s : [p_1, \ldots, p_n]], E) = \emptyset \cup \bigcup_{1 \leq k \leq m} En^*(p_k, E).
\]

For an AND-statechart, variables are shared by all orthogonal components. However, each variable can only be modified by one component. We use \( W \text{Var}(p) \) 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 \( E = \{a_1, b, c\} \), which will fire both \( \tau_3 \) and \( \tau_4 \) 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 \( t_1, \ldots, t_n \) is a permutation of \( 1, \ldots, n \).

\[
p = [[s : [p_1, \ldots, p_n]], \text{all } p_i \text{ are constructed by Basic or Or}}\]

\[
\langle p_{[i_1 \rightarrow i_k]}, E \rangle \xrightarrow{\tau_{[i_k \rightarrow i_k]}} \langle p_{[i_1 \rightarrow i_k]}, \text{\sigma}_{[i_1 \rightarrow i_k]}, E \rangle, \text{for all } 1 \leq k \leq m
\]

\[
En^*(p_i, E) = \emptyset, \text{for all } m < k \leq n
\]

\[
W \text{Var}(p) \cap W \text{Var}(p_i) = \emptyset, \text{for all } i, j, \text{where } i \neq j
\]

\[
\sigma^* = \sigma_{[i_1 \rightarrow i_k]} \cdots \sigma_{[i_{m+1} \rightarrow i_{m+n}]}
\]

\[
E' = \langle E - \bigcup_{1 \leq k \leq m} \text{trig}^+(\tau_{[i_k]}) \rangle \cup \bigcup_{1 \leq k \leq m} \alpha^*(\tau_{[i_k]}), E' \rangle
\]

\[
\langle \hat{p} = [s : [p_{[i_1 \rightarrow i_k]}, p_{[i_{m+1} \rightarrow i_{m+n}]}]] \rangle
\]

\[
\langle p, \sigma, E \rangle \xrightarrow{\alpha^*} \langle \hat{p}, \sigma', E' \rangle
\]

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

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

\[
En^*(p_i, E) = \emptyset, i = 1, \ldots, n
\]

\[
\langle p = [[s : [p_1, \ldots, p_n]], \sigma, E] \xrightarrow{\gamma} \langle p, \sigma, \emptyset \rangle
\]

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

\[ P := \text{seq} \mid P \parallel P \mid \text{var } x \cdot P \]

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

\[ S := \text{PC}(\text{primitive command}) \mid S ; S \text{ (sequential composition)} \mid S \parallel b > S \text{ (conditional)} \mid b \star S \text{ (iteration)} \mid (g S) \ldots (g S) \text{ (guarded choice)} \mid \text{always } S \text{ (infinite loop)} \mid \text{case } (e) (p t S) \ldots (p t S) \text{ (switch statement)} \]

where

\[ \text{PC} := v := e \mid \text{sink} \mid \text{skip} \mid \bot \mid \Rightarrow \eta \mid v := eg e \]

\[ g := \#n \text{ (time delay)} \mid eg \text{ (event control)} \mid \Rightarrow \eta \text{ (output event)} \]

\[ eg := \#n \mid eg \mid eg \& eg \mid eg \& \neg eg \]

\[ \eta := \Downarrow v \text{ (value change)} \mid \Uparrow v \text{ (value falling)} \mid \ conform \text{ (a set of abstract events)} \]

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 \( P \cap Q \)

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 above-mentioned 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.
- \( \sigma : \text{Var} \rightarrow \text{Val} \) 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 \( \sqrt{\cdot} \), or a compositional event possibly with three conjunctive parts: \( b \& g' \& g'' \) 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.

\[
\langle \text{sink}, \sigma, E \rangle \xrightarrow{\text{clock tick}} \langle \text{sink}, \sigma, \emptyset \rangle
\]

The guarded choice construct

\[
P = (b_1 \& g_1 \wedge P_1) \ldots (b_n \& g_n \wedge P_n)
\]

can take a guarded transition if that guard is enabled.

\[
\langle P, \sigma, E \rangle \xrightarrow{b_k \& g_k} \langle P_{k'}, \sigma', E - e^c(g_k) \cup e^a(g_k) \rangle
\]

where \(E \vdash g^c\) indicates that the input guard \(g^c\) is enabled by \(E\). This is defined as:

\[
E \vdash \{e_1 \cap \ldots \cap e_m \} \rightarrow \{e_1 \cup \ldots \cup e_m \} = df \\
\land 1 \leq i \leq m (e_i \subseteq E) \land \land 1 \leq i \leq m (e_i \cap E = \emptyset)
\]

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

\[
e^c(g^c) = df \bigcup_{1 \leq i \leq m} e_i
\]

and \(e^a(g^c)\) records the set of events generated by the output guard \(g^a\). Given an output guard \(g^a = \rightarrow e^a(g) (x = v)\), the generated events are

\[
e^a(g^a) = df \begin{cases} \\
\{ v \cup \uparrow x \}, & \text{if } \sigma(x) < v, \\
\{ v \cup \downarrow x \}, & \text{if } \sigma(x) > v,
\end{cases}
\]

and \(\emptyset\), otherwise.

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

\[
\forall k : 1 \leq k \leq n \rightarrow (\sigma(b_k) \wedge (E \vdash g_k))
\]

\[
\langle P, \sigma, E \rangle \xrightarrow{\emptyset} \langle P', \sigma', E' \rangle
\]

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

\[
G_1 \| \ldots \| G_n
\]

where \(G_k = \bigwedge_{1 \leq j \leq n} b_{jk} \& g_{jk} \wedge P_{jk}, 1 \leq k \leq n\)

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 \(I_1, \ldots, I_n\) denotes a permutation of \(1, \ldots, n\).

\[
\langle G_{i_1}, \sigma, E \rangle \xrightarrow{I_{i_k}} \langle P_{i_1}, \sigma_{i_1}, E_{i_1} \rangle, 1 \leq k \leq m
\]

\[
\forall j : 1 \leq j \leq n \rightarrow (\sigma(b_{ij}) \wedge (E \vdash g_{ij}))
\]

\[
E' = (E - \bigcup_{1 \leq k \leq m} e^c(g_{ik})) \cup \bigcup_{1 \leq k \leq m} e^a(g_{ik})
\]

\[
\langle P, \sigma, E \rangle \xrightarrow{E'} \langle P', \sigma', E' \rangle
\]

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

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

\[
\langle P, \sigma, E \rangle \xrightarrow{\text{clock tick}} \langle P', \sigma', E' \rangle
\]

3.2 Algebraic Laws

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

**Definition 1** Given an event control \(egl\), we define those simple events that enable \(egl\) as follows.

\[
\text{tr}(egl) = df \begin{cases} \\
\{ \uparrow & x \}, & \text{if } egl = \uparrow x \\
\{ \downarrow & x \}, & \text{if } egl = \downarrow x \\
\{ \uparrow & \downarrow x \}, & \text{if } egl = \approx x
\end{cases}
\]

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

\[
\eta \rightsquigarrow egl = df \text{tr}(\eta) \cap \text{tr}(egl) = \emptyset
\]

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

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

\[\square\]
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.

\[
\text{(assign-1)} \quad v := e; \quad \text{if } v := f = v := f[e/v] 
\]

In an assignment to a list of variables, the order of variables is irrelevant.

\[
\text{(assign-2)} \quad u, v := e, f = v, u := f, e 
\]

Variables not occurred on the left side of an assignment remain unchanged during the assignment.

\[
\text{(assign-3)} \quad u := e = u, v := e, v 
\]

`skip` does not change the value of any variable.

\[
\text{(assign-4)} \quad \text{skip } = v := v 
\]

Sequential composition is associative, and has left zero \( \bot \). It distributes backward over conditional, internal and external choices.

\[
\text{(seq-1)} \quad (P; Q); R = P; (Q; R) 
\]

\[
\text{(seq-2)} \quad \bot; P = \bot 
\]

\[
\text{(seq-3)} \quad (P \sqcap Q); R = (P; R) \sqcap (Q; R) 
\]

\[
\text{(seq-4)} \quad (P \sqsubset b \triangleright Q); R = (P; R) \sqsubset b \triangleright (Q; R) 
\]

\[
\text{(seq-5)} \quad \{ l \in I \mid (g_l; q_i) \}; R = \{ l \in I \mid (g_l; q_i; R) \} 
\]

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 \( P \).

\[
\text{(seq-6)} \quad \text{Let } S = \{ l \in I \mid (g_l; P_l) \}, \text{ and } g \text{ is the disjunction of all input guards of } S. 

\[
(1) \quad \rightarrow \eta S = \{ \rightarrow \eta S \text{ if } \eta \rightarrow g; \rightarrow \eta P_k \text{ if } \eta \rightarrow g_k \text{ for some } k \in I. 
\]

\[
(2) \quad (x < f) \bot; \{ (x := f) \}; S = \{ (x < f) \bot; \{ (x := f) \}; g \text{ if } x \rightarrow g; \{ (x < f) \bot; \{ (x := f) \}; P_k \text{ if } x \rightarrow g \text{ for some } k \in I. 
\]

\[
(3) \quad (x > f) \bot; \{ (x := f) \}; S = \{ (x > f) \bot; \{ (x := f) \}; g \text{ if } x \rightarrow g; \{ (x > f) \bot; \{ (x := f) \}; P_k \text{ if } x \rightarrow g \text{ for some } k \in I. 
\]

\[
(4) \quad (x = f) \bot; \{ (x := f) \}; S = \{ (x = f) \bot; \{ (x := f) \}; x \} 
\]

where \( b \bot \) is an assertion defined as \( \text{skip } \sqsubset b \triangleright \bot \) \([14]\).

For a general guarded choice \( G \), we can also transform it by this law into a guarded choice \( l \in I \mid (g_l; P_l) \), where no output guard in \( \{ g_l \mid l \in I \} \) 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.

\[
\text{(cond-1)} \quad v := e; (P \sqsubset b(v) \triangleright Q) = (v := e; P) \sqsubset b(e) \triangleright (v := e; Q) 
\]

Iteration is subject to the fixed point theorem.

\[
\text{(iter-1)} \quad b * P = (P; b * P) \sqsubset b \triangleright \text{skip} 
\]

Non-deterministic choice is idempotent, symmetric and associative.

\[
\text{(nond-1)} \quad P \sqcap P = P 
\]

\[
\text{(nond-2)} \quad P \sqcap Q = Q \sqcap P 
\]

\[
\text{(nond-3)} \quad P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R 
\]

Parallel operator is symmetric and associative, and has \( \bot \) as zero.

\[
\text{(par-1)} \quad P \parallel Q = Q \parallel P 
\]

\[
\text{(par-2)} \quad P \parallel (Q \parallel R) = (P \parallel Q) \parallel R 
\]

\[
\text{(par-3)} \quad \bot \parallel P = \bot 
\]

Local variable declaration enjoys the following laws.

\[
\text{(var-1)} \quad \text{var } x \cdot (x := e) = \text{skip} 
\]

\[
\text{(var-2)} \quad \text{var } x \cdot (P \sqsubset b \triangleright Q) = \text{var } x \cdot P \sqsubset b \triangleright \text{var } x \cdot Q, \quad \text{provided } P \text{ is not free in } b. 
\]

\[
\text{(var-3)} \quad \text{If } x \text{ is not free in } Q, \text{ then} 

\quad (1) \quad \text{var } x \cdot Q = Q 
\quad (2) \quad \text{var } x \cdot (P; Q) = \text{var } x \cdot (P; Q) 
\quad (3) \quad Q; \text{var } x \cdot (P; Q) = \text{var } x \cdot Q; P 
\quad (4) \quad \text{var } x \cdot P \parallel Q = \text{var } x \cdot (P \parallel Q) 
\]

\[
\text{(var-4)} \quad \text{var } x \cdot \rightarrow \eta P = \text{var } v \cdot \text{skip; } P 
\]

\[
\text{(var-5)} \quad \text{var } u \cdot \text{var } v \cdot P = \text{var } v \cdot \text{var } u \cdot P 
\]

We will denote \( \text{var } x \cdot \text{var } y \cdot \ldots \cdot \text{var } x \text{ as } \text{var } x, y, \ldots, z. \)

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

\[
G_1 = \{ l \in I \mid (g_l; Q_l) \} 
\]

\[
G_2 = \{ l \in J \mid (h_j; R_j) \} 
\]

\[
G_3 = \{ l \in K \mid (e_v; P_k) \} 
\]

where all \( g_l \) and \( h_j \) are input guards (like \( \eta \)); all \( e_v \) and \( e_u \) are respectively output events with respect to variables \( v_k \) and \( u_l \) (like \( \eta \) or \( \eta ((x := f)) \)).

\[
\text{(par-1)} \quad (x := e; G_1) \parallel (y := f; G_2) = \begin{align*} 
& (\text{at } x := e)(G_1 \parallel (y := f; G_2)) \mid \text{at } (y := f)(x := e; G_1) \parallel G_2) 
& \end{align*} 
\]

\[
\text{(par-2)} \quad G_1 \parallel (y := f; G_2) = \begin{align*} 
& (\text{at } y := f)(G_1 \parallel G_2) \mid \text{at } (y := f; G_2) 
& \end{align*} 
\]

\[
\text{(par-3)} \quad G_1 \parallel G_2 = \begin{align*} 
& \text{at } (y := f; G_2) 
& \end{align*} 
\]

\[
\text{Let } g = g_l \lor \text{or } i \in I \quad g_i, h = g_l \lor \text{or } j \in J \quad h_j, \text{ then} 

\[ (G_1 \parallel G_2) \parallel (G_3 \parallel G_4) = \{ l \in I \mid (g_l \lor \neg h_l) \parallel (Q_l \parallel (G_2 \parallel G_4)) \} \]
\[
\begin{align*}
\{ j \in J \mid (h_j &\land \neg g) ((G_1 \parallel G_3) \parallel R_j) \} \\
\{ j \in J \mid (g_j &\land h_j) (Q_1 \parallel R_j) \} \\
\{ k \in K \mid (r_k \rightarrow h_j (e_{v_k} (P_k \parallel G_q))) \} \\
\{ k \in K, r_k \rightarrow h_j (e_{v_k} (P_k \parallel G_q)) \} \\
\{ i \in I, j \in J \mid (g_i \land h_j) (Q_i \parallel R_j) \} \\
\{ i \in I, j \in J \mid (g_i \land h_j) (Q_i \parallel R_j) \} \\
\{ i \in I, j \in J \mid (g_i \land h_j) (Q_i \parallel R_j) \} \\
\{ i \in I, j \in J \mid (g_i \land h_j) (Q_i \parallel R_j) \} \\
\{ i \in I, j \in J \mid (g_i \land h_j) (Q_i \parallel R_j) \} \\
\{ i \in I, j \in J \mid (g_i \land h_j) (Q_i \parallel R_j) \} \\
\end{align*}
\]

(par-7) An assignment thread is involved.

1. \((x := e) \parallel (y := e) = (\alpha(x := e), y := e)) \parallel (\alpha(y := e), x := e))\)

2. \((x := e) \parallel G_2 = (\alpha(x := e), G_2) \parallel \{ j \in J \mid (h_j ((x := e) \parallel R_j))\} \)

The parallel operator is disjunctive.

(par-8) \((P \parallel Q) \parallel R = (P \parallel R) \parallel (Q \parallel R)\)

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

(par-9) \[\text{var} v_1, \ldots, v_n \subseteq \{ S_1 < b \parallel S_2 > G \} = \text{var} v_1, \ldots, v_n \subseteq \{ S_1 > b \parallel (S_2 > G)\},\]

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. \((\#n S) \parallel G_3 = G_3\)

2. \((G_1 \#n_1 S) \parallel (G_2 \#n_2 T) = \left[ \begin{array}{l} \{ i \in I \mid (g_i \land \neg h) (Q_1 \parallel (G_2 \#n_2 T)) \} \\
\{ j \in J \mid (h_j \#n_1 S) \parallel R_j \} \} \\
\{ i \in I, j \in J \mid (g_i \land h_j) (Q_i \parallel R_j) \} \} \\
\#n_2 ((\#n_1 - n_2) S \parallel T) \}
\end{array} \right)

3. \((G_1 \#n S) \parallel G_2 \#n T = \left[ \begin{array}{l} \{ i \in I \mid (g_i \land \neg h) (Q_i \parallel (G_1 \#n S)) \} \\
\{ j \in J \mid (h_j \#n S) \parallel R_j \} \} \\
\{ i \in I \mid (g_i \land h_j) (Q_i \parallel R_j) \} \\
\#n ((S \parallel T) \}
\end{array} \right)
\]

The guarded choice is idempotent, symmetric and associative.

(guard-1) \(G_1 \parallel G_1 = G_1\)

(guard-2) \(G_1 \parallel G_2 = G_2 \parallel G_1\)

(guard-3) \((g_1 Q_1) \parallel ((g_2 Q_2) \parallel (g_3 Q_3)) = (g_1 Q_1) \parallel (g_2 Q_2) \parallel (g_3 Q_3)\)

(guard-4) \(\text{var} v \subseteq (\#n P) \parallel G_1 = \text{var} v \subseteq G_1\)

The construct always \(S\) executes \(S\) forever.

(always-1) \(always S = S; always S\)

Take note that \(\text{skip} = \text{not a left zero of sequential composition in general cases, because it might filter some signal. Hereby, the following in-equation is obvious.}\)

\[\uparrow v \neq \text{skip}; \uparrow v\]

The following definition will capture those cases where \(\text{skip}\) is a left zero of sequential composition.

**Definition 3 (Event control insensitive)**

A process \(P\) is event control insensitive if \(\text{skip}_P = P\).

\(\square\)

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

1. \(x \rightarrow e, \text{skip}, \perp, \text{or } \#(t)\),
2. \(\alpha(x := e), \rightarrow \eta_k\),
3. \(P < b \parallel Q, b \parallel Q, \text{case } (e) (p_h \parallel S) \ldots (p_t \parallel S_n)\),
4. \(\text{var} v_1, \ldots, v_n \subseteq (S_1 \parallel \ldots \parallel S_n), \text{where each } S_i \text{ is either event control sensitive, or only guarded by events with respect to variables in } \{v_1, \ldots, v_n\}\).

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

**Lemma 1** Let \(P = (\eta_p P_2), Q = (\eta_q Q_2)\),

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. \(\text{var} u, v \subseteq (P \parallel Q) \equiv \text{var} u, v \subseteq (P_2 \parallel (\eta_q Q_2))\)
2. \(\text{var} u, v \subseteq (P \parallel (Q_1; Q)) \equiv \text{var} u, v \subseteq (Q_1; (P \parallel Q))\)
3. \(\text{var} u, v \subseteq ((P_1; P) \parallel (Q_1; Q)) \equiv \text{var} u, v \subseteq ((P_1 \parallel Q_1); (P \parallel Q))\)

\(\square\)

**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 \(P \subseteq Q\), if \(P \parallel Q = P\) is algebraically provable.

\(\square\)

## 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 subtrees. By applying 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: $\text{SC} \to \mathbb{N}$ to calculate the “or-depth” of a statechart, which is defined as follows:

- for a statechart $c = [s]$ constructed by Basic, $\text{or-depth}(c) = 0$;
- for a statechart $c = [s : \{p_1, \cdots, p_m, p,T\}]$ constructed by Or, $\text{or-depth}(c) = \text{or-depth}(p) + 1$;
- for a statechart $c = [s : \{p_1, \cdots, p_n\}]$ constructed by And, $\text{or-depth}(c) = 1$.

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 $c = [s : \{p_1, \cdots, p_m, p,T\}]$, $\text{active}(c) = p$ denotes its current active state; for any transition $\tau \in T$, $\text{src}(\tau)$ and $\text{tg}(\tau)$ 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 active constitutents, i.e., $\text{active}(c) = \{\text{active}(p_1), \cdots, \text{active}(p_n)\}$. Use $T(c)$ to denote all possible (perhaps compositional) transitions of the And-chart $c$. Given a transition $\tau = \&_{1\leq k\leq n} \tau_k \in T(c)$, where $\tau_k \in T^*(\text{active}(p_k))$, for $1 \leq k \leq m$, and $i_1, \cdots, i_n$ is a permutation of $1, \cdots, n$, we define its source state and target state respectively as follows:

\[
\text{src}(\tau) = q_{i_1} \cdots q_{i_n}, \quad \text{tg}(\tau) = q_{i_1} \cdots q_{i_n},
\]

for $1 \leq k \leq m$, and $q_{i_k} = \text{active}(p_{i_k})$, for $m < k \leq n$.

For an Or-chart $p = [s : \{p_1, \cdots, p_m, p,T\}]$, $T(p)$ contains all possible transitions inside $p$ along its transitive active subtree chain, i.e., $T(p) = \{\tau \mid \tau \in T \land \text{src}(\tau) = p\} \cup T(p)$.

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 $T(c) = \{\tau \mid \tau \in T^*(p_i), i = 1, 2 \cup \{\tau_{i_1} \cdots \tau_{i_N} \mid \tau_{i_j} \in T(p_{i_j}), i = 1, 2\} \land \text{src}(\tau_{i_1} \cdots \tau_{i_N}) = q_{i_1} \cdots q_{i_N}, {1 \leq k \leq m, \text{active}(p_{i_k})\}}$.

The transition set for the general And-chart with $n$ components can be defined similarly.

\[
\text{tg}(\tau) = \{r_1, \cdots, r_n\}, \quad \text{where} \quad r_{i_k} = \text{tg}(\text{active}(p_{i_k})), \quad \text{for} \quad 1 \leq k \leq m, \text{and} \quad r_{i_k} = \text{active}(p_{i_k}), \quad \text{for} \quad m < k \leq n.
\]

Thirdly, we need to know the resulting statechart after a transition is taken. When a transition $\tau$ occurs, any involved statechart can have changes in its (transitive) active subtrees. We use a function $\text{resc}: T \times \text{SC} \to \text{SC}$ 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 $\tau$, $\text{resc}(\tau,c) = \{c\}$;
- For an Or-chart $c = [s : \{p_1, \cdots, p_m, p,T\}]$, and a transition $\tau$,

\[
\text{resc}(\tau,c) = \{c_{\text{active}(p_i)} \} \quad \text{if} \quad \tau \in T \land \text{src}(\tau) = p_i,
\]

\[
\text{resc}(\tau,c) = \{c_i, \text{otherwise} \}
\]

- For an And-chart $c = [s : \{p_1, \cdots, p_n\}]$, and a transition $\tau$,

\[
\text{resc}(\tau,c) = \{c_i \mid \tau = \&_{1\leq k\leq m} \tau_k \in T(c),
\]

\[
\text{active}(c) = \{\text{active}(p_{i_1}), \cdots, \text{active}(p_{i_n})\}
\]

where $c_i = \{q_{i_1}/p_{i_1}, \cdots, q_{i_n}/p_{i_n}\}$ is the statechart obtained from $c$ via replacing $p_i$ by $q_i$ for $1 \leq i \leq n$, $q_{i_k} = \text{resc}(\text{active}(p_{i_k}), p_{i_k})$, for $1 \leq k \leq m$, and $q_{i_k} = p_{i_k}$, for $m < k \leq n$.

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 $L: \text{SC} \to \text{Verilog}$ maps any statechart description into a corresponding Verilog process. It keeps unchanged the set of variables employed by the source description, i.e.,

\[
\forall c \in \text{SC} \cdots \text{vars}(L(c)) = \text{vars}(c)
\]

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.,

\[
L(c) = \text{sink}
\]

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

\[
L(c) = \begin{cases} \text{sink}, & \text{if } T^*(c) = \emptyset \\ P, & \text{otherwise} \end{cases}
\]

where

\[
P = \text{if } \sum_{(c)} \text{or-depth}(c), \quad \text{then } \begin{cases} I_{\text{sink}}, & \text{if } c_i \in T^*(\text{active}(c)) \land \text{src}(\tau_k) = \text{active}(c) \land h_i = \{\tau \in T^*(\text{active}(c)) \land \text{src}(\tau_k) = \text{active}(c) \}
\end{cases}
\]

For a statechart $c = [s : \{p_1, \cdots, p_m, p,T\}]$ constructed by Or, we define $L$ by exhaustively figuring out the first possible transitions of $c$ if any, otherwise it sinks.

\[
L(c) = \begin{cases} \text{sink}, & \text{if } T^*(c) = \emptyset \\ P, & \text{otherwise} \end{cases}
\]

where

\[
P = \begin{cases} \text{sink}, & \text{if } c_i \in T^*(\text{active}(c)) \land \text{src}(\tau_k) = \text{active}(c) \land h_i = \{\tau \in T^*(\text{active}(c)) \land \text{src}(\tau_k) = \text{active}(c) \}
\end{cases}
\]
and

\[ \text{active}^0(c) = \delta \text{, active}^1(c) = \delta \text{, active}(c) \]

\[ \text{active}^{i+1}(c) = \gamma \text{ active}(c) \]

The input guard \( g_1 \) comprises the overall trigger events of \( \tau_k \), which has the form \( \xi_1 \& \xi_2 \), where \( \xi_1 \) are events from \( \text{trig}^\text{in}(\tau_k) \), whereas \( \xi_2 \) are events out of \( \text{trig}^\text{out}(\tau_k) \).

Due to the priority mechanism of Statecharts, an enabled transition \( \tau_k \) in an inner level \( (k) \) can occur only when no transitions from any outer level \((0, \ldots, k-1)\) are enabled. The part \( (\&_{1 \leq j \leq k} h_j) \) is used to denote this condition.

The output guard \( g_o \) is the overall action performed by \( \tau_k \), which has the form \( \rightarrow \xi \& \text{out}(x = v) \), where \( \xi \) comprises all abstract events out of \( a^r(\tau_k) \), and the assignment action \( x = v \) is from \( a^d(\tau_k) \).

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:

\[ p = \left[ a : \{ p_1, p_2 \}, p_1, \{ t_1, t_2 \} \right] \]

where \( t_i = \delta (p_i, \{ e \}, \emptyset, \text{true}, p_{i-1}) \), \( i = 1, 2 \).

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:

\[ q = \left[ a : \{ q_1, q_2 \} \right] \]

\[ q_1 = \left[ a : \{ q_3, q_4 \}, q_3, \{ t_3 \} \right] \]

\[ q_2 = \left[ a : \{ q_5, q_6 \}, q_5, \{ t_4 \} \right] \]

where

\[ t_3 = (q_3, \{ a, b \}, \{ e \}, \text{true}, p_4) \]

\[ t_4 = (q_5, \{ f \}, \{ g \}, \text{true}, p_5) \]

It is mapped into the following parallel construct

\[ \text{(a&b&c&d&e&f&sink) || ((e&((e&g&sink)(b&c&d&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:

\[ p = \left[ a : \{ p_1, p_0 \}, p_1, \{ t_1 \} \right] \]

\[ p_1 = \left[ a : \{ p_2, p_0 \}, p_2, \{ t_2, t_3 \} \right] \]

\[ p_2 = \left[ a : \{ p_3, p_0 \}, p_3, \{ t_4 \} \right] \]

\[ p_3 = \left[ a : \{ p_5, p_0 \}, p_5, \{ t_6 \} \right] \]

\[ p_4 = \left[ a : \{ p_7, p_6 \}, p_7, \{ t_8 \} \right] \]

\[ p_{10} = \left[ a : \{ p_1, p_2 \}, p_1, \{ t_6, t_7 \} \right] \]

**Fig. 3** A more complicated statechart

where

\[ t_1 = (p_1, \{ e \}, \{ @ (y = 0) \}, \text{true}, p_{10}) \]

\[ t_2 = (p_0, \emptyset, \emptyset, x > 0, p_2) \]

\[ t_3 = (p_2, \{ d \}, \{ @ (x = x - 1) \}, \text{true}, p_0) \]

\[ t_4 = (p_0, \{ b \}, \{ e \}, \text{true}, p_0) \]

\[ t_5 = (p_7, \{ a \}, \emptyset, \text{true}, p_8) \]

\[ t_6 = (p_{12}, \emptyset, \emptyset, y < 10, p_{11}) \]

\[ t_7 = (p_{11}, \{ f \}, \{ @ (y = y + 1) \}, \text{true}, p_{20}) \]

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

\[ \mu X \bullet \left( \begin{array}{c} Q \mid P \\ \left( (x > 0) \& @ (y = y + 1) \right) \mid \left( (x > 0) \& @ (y = y + 1) \right) \end{array} \right) \]

where

\[ Q = \delta (e \& @ (y = 0)) \]

\[ P = \delta (f \& @ (y = y + 1)) \]

\[ (x > 0) \& @ (y = y + 1) \]

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 \( \text{Fan} \), \( \text{Temperature} \), \( \text{Timer} \), \( \text{TempDisplay} \), and \( \text{TimerDisplay} \). They will be respectively mapped to Verilog programs \( p\text{Fan} \), \( p\text{Temperature} \), \( p\text{Timer} \), \( p\text{TempDisplay} \), and \( p\text{TimerDisplay} \).

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

\[ pon = \delta \mu X \bullet \left( \begin{array}{c} p\text{Fan} \\ p\text{Temperature} \\ p\text{Timer} \\ p\text{TempDisplay} \\ p\text{TimerDisplay} \end{array} \right) \]

The five component programs are respectively

\[ p\text{Fan} = \delta \mu X \bullet \left( \begin{array}{c} f\text{Fan} \left( f\text{Fan} (f\text{Fan} X) \right) \end{array} \right) \]

\[ p\text{Temperature} = \delta \mu X \bullet \left( \begin{array}{c} (v < 28) \& e\text{Incr} \& @ (v = v + 1) \\ (v < 16) \& e\text{Decr} \& @ (v = v - 1) \end{array} \right) \]

\[ p\text{Timer} = \delta \mu X \bullet \left( \begin{array}{c} (\text{bit}\text{timer} \& \rightarrow \text{timeron}) \mid P \end{array} \right) \]
where
\[ P = \delta(Y) \cdot \begin{cases} (t < 8) \& h\text{Incr} \& \lnot t + 1 \to Y \\ (t > 1) \& h\text{Decr} \& \lnot t - 1 \to Y \\ (t > 1) \& h\text{Decr} \& \lnot t - 1 \to Y \\ (t > 1) \& h\text{Decr} \& \lnot t - 1 \to Y \\ (t > 1) \& h\text{Decr} \& \lnot t - 1 \to Y \\ (t > 1) \& h\text{Decr} \& \lnot t - 1 \to Y \\ (t > 1) \& h\text{Decr} \& \lnot t - 1 \to Y \\ \end{cases} \]

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 \( \tau \) which leads to statechart \( C' \), there exists a Verilog transition \( t \) 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.

**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 \( L(C) = \text{sink} \), which does nothing but performs the clock tick and remains as \( \text{sink} \) after that.

2. \( C = [s: [p_1, \ldots, p_n], p_1, T] \) 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 \( \tau \in T(\text{active}(C)) \) for some \( k \geq 0 \) in case that all transitions of outer levels (if any) are not available, which changes the active state of \( \text{active}^k(C) \) from its source state to its target state and results in \( \text{new}(\tau, C) \); (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 \( \{ (g, P) \} \). If (1) occurs, \( g \), is fired, from the semantics of Verilog, such a program can perform the corresponding transition and become \( P' \), otherwise it can perform the clock tick transition. From the definition of \( L \), it is straightforward that \( P' = L(\text{resc}(\tau, C)) \).
   The second part can be proved similarly from the definition of \( L \).

3. \( C = [s: [p_1, \ldots, p_n]] \) is constructed by And.
   From Definition 5, we know
   \[ L(C) = L(p_1) \cdot \cdots \cdot L(p_n) \]
   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 \( \tau \) can be performed at the current environment, from rule 5, we know that \( \tau_k \), for \( 1 \leq k \leq m \), are ready to take place and orthogonal components other than \( p_1, \ldots, p_m \) do not have available transitions. This implies all processes \( L(p_1), \ldots, L(p_m) \) can take the transition corresponding to \( \tau_1, \ldots, \tau_m \) respectively in the current environment, whereas oth-
ers can not. From the operational semantics of parallel construct of Verilog, a parallel transition corresponding to \( \tau \) can take place and after the transition the program becomes
\[
P_1 \parallel \cdots \parallel P_n
\]
where
\[
P_i = \begin{cases} 
L(\text{resc}(\tau_i, p_i)), & \text{for } 1 \leq i \leq m, \\
L(p_i), & \text{otherwise.}
\end{cases}
\]
It exactly accords with \( L(\text{resc}(\tau, C)) \). 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 \( \sigma \) 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 \( L \). 

\[ \Box \]

## 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.

\[
S ::= AC \quad (\text{primitive command}) \\
| S; S \quad (\text{sequential composition}) \\
| S \mid b \mid S \quad (\text{conditional}) \\
| S \mid S \quad (\text{non-deterministic choice}) \\
| b \ast S \quad (\text{iteration}) \\
| (gS) \mid (g S) \quad (\text{guarded choice})
\]

where

\[
AC ::= PC \quad (\text{defined in section 3}) \\
| (v := e) \quad (\text{timing assignment}) \\
| \langle S \rangle \quad (\text{specific block})
\]

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
\[
(v := e)_n = \bigvee_{0 \leq k \leq n} (v := \#k e)
\]
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
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. $\Rightarrow \eta_k; C; \eta_a$, where $C$ is a member of $CP(r, a)$ not mentioning $r, a$;
3. $C_1; C_2$, or $C_1 \triangleleft C_2$, or $C_1 \cap C_2$, or $(g_1; C_1) \mid (g_2; C_2)$, where $C_1, C_2, g_1, g_2 \in CP(r, a)$;
4. $b \star C$, where $C \in CP(r, a)$.

We introduce another set $CP_e(r, a)$ comprising those processes in $CP(r, a)$ not mentioning variable $r$.

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 $(C; \Rightarrow \eta_k), C$ is a member of $CP_e(r, a)$, the special event $\Rightarrow \eta_k$ is adopted for the purpose of synchronization between hardware and software, and the hardware component $D_0$ is subject to the following equation:

$$D_0 = \mu X \bullet \{(\eta_a; M; \Rightarrow \eta_k; D_0) \parallel (\eta_k; \text{skip})\}$$

where

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

is a case construct not containing $r, a, \varepsilon$.

We denote as $D_P(r, a)$ 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

$$\Psi_f(S) =_{df} \text{always}(\eta_k; S; \Rightarrow \eta_f)$$

which is driven by the environmental process:

$$Env =_{df} \text{always}(\Rightarrow \eta_k; \eta_f)$$

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

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

where

$$P \parallel_{Env} Q =_{df} P \parallel Env \parallel Q$$
The software component enjoys the form
\[ \Psi_j(C) = \delta \text{ always } (\eta_j C; \rightarrow \eta_j) \]
where \( C \) is a process from \( CP_r(a) \); the hardware component \( D \) is of the form:
\[ D = \delta \text{ always } (\eta_j M; \rightarrow \eta_j) \]

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
\[ (C_1; C_2; \rightarrow \eta_j) \mid D_0 = ((C_1; \rightarrow \eta_j) \mid D_0); ((C_2; \rightarrow \eta_j) \mid D_0) \]
for any \( C_1, C_2 \in CP_r(a) \) and \( D_0 \in DP_r(a) \).

**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.
\[ C_1 = \Delta, \text{ the proof is trivial.} \]
\[ C_1 = \delta g, \text{ where } g \text{ is } \alpha(x := e) \text{ or } \eta_j \text{ or } \#n. \]
\[ LHS = (\delta g; (C_2; \rightarrow \eta_j) \mid D_0) \]
\[ \{ (\text{seq-0}), (\text{par-0}), (\text{guard-0}) \} \]
\[ \text{Proposition 1} \]
\[ = \delta g; (\text{skip}; (C_2; \rightarrow \eta_j) \mid D_0) \]
\[ \{ (\text{par-0}), (\text{ivar-0}) \} \]
\[ = \delta g; (\text{skip}; (C_2; \rightarrow \eta_j) \mid D_0) \]
\[ \{ (\text{seq-0}), (\text{par-0}), (\text{guard-0}) \} \]
\[ \text{RHS} \]

(1.2) \( C_1 = S_1 \cap S_2 \)
\[ LHS = ((S_1; C_2; \rightarrow \eta_j) \mid D_0) \]
\[ \{ (\text{seq-0}) \} \]
\[ = ((S_1; C_2; \rightarrow \eta_j) \mid D_0) \]
\[ \{ (\text{par-0}) \} \]
\[ = (\delta g; (\text{skip}; (C_2; \rightarrow \eta_j) \mid D_0)) \]
\[ \{ (\text{ivar-0}) \} \]
\[ = (\delta g; (\text{skip}; (C_2; \rightarrow \eta_j) \mid D_0)) \]
\[ \{ (\text{seq-0}) \} \]
\[ \text{RHS} \]

(1.3) \( C_1 = S_1 \cup b \triangleright S_2 \)
\[ LHS = (\delta g; (\text{skip}; (S_2; C_2; \rightarrow \eta_j)) \mid D_0) \]
\[ \{ (\text{seq-0}) \} \]
\[ = (\delta g; (\text{skip}; (S_2; C_2; \rightarrow \eta_j)) \mid D_0) \]
\[ \{ (\text{par-0}) \} \]
\[ = (\delta g; (\text{skip}; (S_2; C_2; \rightarrow \eta_j)) \mid D_0) \]
\[ \{ (\text{seq-0}) \} \]
\[ \text{RHS} \]

(1.4) \( C_1 = \llcorner g(E, S_1) \)
\[ LHS = (\delta g(E, S_1) \mid D_0) \]
\[ \{ (\text{seq-0}) \} \]
\[ = (\delta g(E, S_1) \mid D_0) \]
\[ \{ (\text{par-0}), (\text{guard-0}) \} \]
\[ \text{hypothesis} \]

(1.5) \( C_1 = b \ast S \)
Let \( F(X) = \delta f (S; X) \text{ skip}_p \) and define \( F^n(\perp) = \delta f \perp_\triangleleft \text{ and } F^{n+1}(\perp) = \delta f F^n(\perp) \), for \( n \geq 0 \).

Then
\[ LHS = (\delta f((F^n(\perp); C_2; \rightarrow \eta_j)) \mid D_0 \}
\[ \{ \text{is continuous} \}
\[ = \bigcup_{n \geq 0} (F^n(\perp); (C_2; \rightarrow \eta_j)) \mid D_0 \}
\[ \{ \text{is continuous} \}
\[ = \bigcup_{n > 0} (F^n(\perp); (C_2; \rightarrow \eta_j)) \mid D_0 \}
\[ \{ \text{hypothesis} \}
\[ = \bigcup_{n > 0} (F^n(\perp); (C_2; \rightarrow \eta_j)) \mid D_0 \}
\[ \{ \text{and ; are continuous} \}
\[ \text{RHS} \]

(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 \( (\text{seq-3}) \), \( (\text{seq-5}) \) and \( (\text{seq-0}) \), 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 = (\delta tg; (S_2; C_2; \rightarrow \eta_j) \mid D_0 \}
\[ \{ \text{hypothesis} \}
\[ = (\delta tg; (S_2; C_2; \rightarrow \eta_j) \mid D_0 \}
\[ \{ \text{hypothesis} \}
\[ = (\delta tg; ((S_2; C_2; \rightarrow \eta_j) \mid D_0) \}
\[ \{ \text{seq-0}, (\text{par-0}), (\text{guard-0}) \} \]
\[ \text{RHS} \]

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

**case 2** \( C_1 = \rightarrow \eta_j; C; \eta_j \).
\[ LHS = (\delta \eta_j; (S_1; C; \rightarrow \eta_j) \mid D_0 \}
\[ \{ \text{par-0}, (\text{ivar-0}) \}
\[ = (\delta \eta_j; (S_1; C; \rightarrow \eta_j) \mid D_0 \}
\[ \{ \text{Lemma 1} \}
\[ = (\delta \eta_j; (S_1; C; \rightarrow \eta_j) \mid D_0 \}
\[ \{ \text{Lemma 1} \}
\[ = (\delta \eta_j; (S_1; C; \rightarrow \eta_j) \mid D_0 \}
\[ \{ \text{Lemma 1} \}
\[ \text{RHS} \]

**case 3** \( C_1 \) is a composite construct.

(3.1) \( C_1 \) is \( C_0 ; C_1 \), similar to (1.6).

(3.2) \( C_1 \) is \( C_0 \cap C_1 \), similar to (1.2).

(3.3) \( C_1 \) is \( C_0 \ast b \triangleright C_1 \), analogous to (1.3).

(3.4) \( C_1 \) is \( \llcorner E_1 (g(E, C_0)) \), similar to (1.4).

(3.5) \( C_1 \) is \( b \ast C_1 \), analogous to (1.5).

The following corollary is directly from theorem 3.

**Corollary 1** Given \( C \in CP_r(a) \) and \( D_0 \in DP_r(a) \), we have
\[ (b \ast C; \rightarrow \eta_j) \mid D_0 = b \ast ((C; \rightarrow \eta_j) \mid D_0) \]
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 into 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 \( \text{Split} \) which plays a vital role in formalizing the splitting rules.

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

\[
\text{Split}_V(S, C, D^0) \equiv 
\begin{cases} 
S \subseteq (C \rightarrow \eta_c) & \text{(|| } D^0) \\
(C \in CP_c(r, a)) \land (D^0 \in D \mathcal{P}_c(r, a)) & \text{(|| } V \in \text{Var}(C \rightarrow \eta_c) \cap \text{Var}(D^0)) \\
V \in \text{Occ Var}(S) = \emptyset 
\end{cases}
\]

where \( \text{Occ Var}(P) \) denotes the set of variables occurred in the program \( P \).

We design two sets 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**

\[
\text{Split}_V(S_i, C_i, D^0), i = 1, 2 \\
\text{Var}(S_1) = \text{Var}(S_2) \\
\text{Split}_V(S_1; S_2, C_1; C_2, D^0)
\]

**Proof** \( S_1 \triangleleft b \triangleright S_2 \) \{ is monotonic \}

\[
\begin{align*}
\subseteq & \ ((C_1; \rightarrow \eta_c) || D_0) \triangleleft (C_2; \rightarrow \eta_c) || D_0) \{ \text{theorem 3} \} \\
= & 
(C_1; \rightarrow \eta_c) || D_0
\end{align*}
\]

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 \( DP_c(r, a) \).

**Definition 7**

\[
D^i \triangleq \mu X \cdot ((\eta_c M^i; \rightarrow \eta_c; X) || (\eta_c \text{ skip}))
\]

where

\[
M^i = \triangleq \text{ case (id) } (p_1 M^i_1) \ldots (p_{n_2} M^i_{n_2}), \text{ for } i = 1, 2
\]

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

\[
\text{mergable}(D^1, D^2)
\]

if

\[
\text{Var}(D^1) = \text{Var}(D^2), \text{ and }
\]

\( p_i = p_j^2 \) 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
\[
D = \text{int}(D_1 \cdot D_2) =_{\text{if}} \mu X \cdot ((\eta; M; \rightarrow \eta; X) \mid (\eta; \text{skip}))
\]
where \( M =_{\text{if}} \text{case} (id) (t_1; M_1), \ldots, (t_r; M_r) \)
and \( \{t_1, \ldots, t_r\} = \{p_{11}, \ldots, p_{1n_1}\} \cup \{p_{21}, \ldots, p_{2n_2}\} \)
and \( M_1, \ldots, M_r = \{M_1, \ldots, M_{n_1}\} \cup \{M_1, \ldots, M_{n_2}\} \).
\]

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**

\[
\text{Split}_V(S, C, D) = \text{merge}_V(D, D')
\]

**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**

\[
\text{Split}_V(S_1; C_1, D_1) = \text{Var}(S_1) = \text{Var}(S_2)
\]

\[
\text{merge}_V(D_1, D_2)
\]

**Top-Down Rule for Conditional**

\[
\text{Split}_V(S_1; C_1, D_1) = \text{Var}(S_1) = \text{Var}(S_2)
\]

\[
\text{merge}_V(D_1, D_2)
\]

**Top-Down Rule for Non-Deterministic Choice**

\[
\text{Split}_V(S_1 \cup S_2, C_1 \cup C_2) = \text{Var}(S_1) = \text{Var}(S_2)
\]

\[
\text{merge}_V(D_1, D_2)
\]

**Top-Down Rule for Guarded Choice**

\[
\text{Split}_V(S_1; C_1, D_1) = \text{Var}(S_1) = \text{Var}(S_2)
\]

\[
\text{merge}_V(D_1, D_2)
\]

\[
\text{Split}_V((g_1; S_1) \mid (g_2; S_2), (g_1; C_1) \mid (g_2; C_2), \text{int}(D_1, D_2))
\]

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 \( (v := f(x, c)_0) \), 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.

\[
\text{Split}_B(S = ((v := f(x, c)_0), C, D), \text{where} \]

\[
C =_{\text{if}} \text{case} (id) (\text{if} := \eta; \eta; (v := \text{ly}_0), \text{and} \]

\[
D =_{\text{if}} \mu X \cdot ((\eta; \text{case} (id) (\text{if} := f(x, c)_0)) \rightarrow \eta; X) \mid (\eta; \text{skip}))
\]

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

\[
\text{Split}_B(S = ((v := f(x, c)_0), C, D), \text{where} \]

\[
C =_{\text{if}} \text{case} (id) (\text{if} := \eta; \eta; (v := \text{ly}_0), \text{and} \]

\[
D =_{\text{if}} \mu X \cdot ((\eta; \text{case} (id) (\text{if} := f(x, c)_0)) \rightarrow \eta; X) \mid (\eta; \text{skip}))
\]

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

\[
\text{Split}_B(S = ((v := f(x, c)_0), C, D), \text{where} \]

\[
C =_{\text{if}} \text{case} (id) (\text{if} := \eta; \eta; (v := f(x, c)_0), \text{and} \]

\[
D =_{\text{if}} \mu X \cdot ((\eta; \text{case} (id) (\text{if} := f(x, c)_0)) \rightarrow \eta; X) \mid (\eta; \text{skip}))
\]

5.2.2 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

\[
\Psi^f(S) =_{\text{if}} \text{always} (\eta_S; \rightarrow \eta_f)
\]

which is driven by environment process

\[
Env =_{\text{if}} \text{always} (\rightarrow \eta_S; \eta_f)
\]

where \( S \) is the kernel specification for the system to be designed, and \( \eta_S \) is the start signal, \( \eta_f \) 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:

\[ \text{Split}_V(S, C, D) \]

where

\[ V = \{ r, a, e, id \}, \]

and \( D = \mu X \cdot ([\eta; M; \rightarrow \eta; X]) (\eta \text{ skip}) \).

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

**System Partitioning Rule**

\[
\begin{align*}
\text{Split}_P(S, C, D) & \equiv \text{Part}(\Psi^P_f(S), \Psi^P_f(C), \Psi^P_f(M)) \\
\text{Part}(S, C, D) & \equiv \bigg( (S \parallel \text{Env}_n) \subseteq (C \parallel D \parallel \text{Env}_n) \\ 
\Psi^P_f(P) & \equiv \text{always}_n(\eta; P; \rightarrow \eta) \\ 
\text{Env}_n & \equiv \text{always}_n(\eta; \eta_f) 
\end{align*}
\]

**Proof** We define \( \{ \text{always}_n(S) \} \) as follows, for all \( n \geq 0 \):

\[ \text{always}_0(S) \equiv \perp, \text{always}_n(S) \equiv \perp S; \text{always}_{n+1}(S) \]

then by law (always-1), we have

\[ \text{always}_n = \bigcup_{n \geq 0} \text{always}_n(S) \]

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

\[
\begin{align*}
\Psi^P_f(S) \parallel \text{Env}_n & \subseteq (\Psi^P_f(C) \parallel \eta) \parallel D \parallel \text{Env}_n \\
\text{Env}_n & \equiv \text{always}_n(\eta; \eta_f)
\end{align*}
\]

By mathematical induction on \( n \).

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

\[
\begin{align*}
\Psi^P_f(S) \parallel \text{Env}_0 & \equiv \perp \subseteq (\perp; \rightarrow \eta) \subseteq (\Psi^P_f(C) \parallel \eta) \parallel D \parallel \text{Env}_0 \\
\text{Env}_0 & \equiv \text{always}_n(\eta; \eta_f)
\end{align*}
\]

2. Inductive step \( (n \rightarrow n+1) \).

We first prove, for all \( n \geq 0 \),

\[
\begin{align*}
\Psi^P_f(C) \parallel \eta & \parallel \text{Env}_n \equiv \text{always}_n(C) \rightarrow \eta \\
\text{Env}_n & \equiv \text{always}_n(\eta; \eta_f)
\end{align*}
\]

By induction on \( n \).

\[ n = 0. \text{ It’s straightforward by law (par-1) and (seq-2).} \]

\[ n \rightarrow n + 1. \]

\[
\begin{align*}
\Psi^P_f(C)_{n+1} \rightarrow \eta & \parallel \text{Env}_{n+1} \\
(\text{par-6}), (\text{par-4}), (\text{Proposition } 1) \\
(\text{Lemma } 1) \\
(\text{Lemma } 2) \\
(\text{Lemma } 3) \\
\text{always}_{n+1}(C) \rightarrow \eta \\
\text{always}_n \rightarrow \eta \\
\text{always}_n \rightarrow \eta
\end{align*}
\]

Then, we have

\[
\Psi^P_f(S)_{n+1} \parallel \text{Env}_{n+1}
\]

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 in [12; 32; 33; 8] are all based on Gordon’s interpretation on simulation cycles. 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]. Beuva et al. [1] and Seshia et al. [27] translate STATEMATE Statecharts to synchronous languages Signal and Estere 1 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 hierarchically 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 ENLOG 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 [23]. 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 refine 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**


Appendix

Proof of Lemma 1

Proof (1). \( LHS = \{par\-6\} \)

\[\begin{align*}
\text{var } u, v & \rightarrow (\eta_u(P) \sqsubset Q) \rightarrow (\eta_v(P) \sqsubset \eta_u(Q)) \\
\text{\{guard-4\}} & \rightarrow (\eta_u(P) \sqsubset \eta_v(Q)) \\
\text{\{par-6\}} & \rightarrow (\eta_u(P) \sqsubset \eta_v(Q)) \\
\text{\{ivar-4\}} & \rightarrow \text{skip}(P) \sqsubset \eta_v(Q) \\
\text{\{Proposition 1\}} & \rightarrow \text{RHS}
\end{align*}\]

(2). By structural induction on \(Q_1\).

(2.1) \(Q_1\) is an atomic command.

It is obvious when \(Q_1 = \text{stop} \) or \(Q_1 = \).

\(Q_1 = t_p\), where \(t_p\) is one of the following forms

\(\ominus(x := e), \eta_{x}, \#n\). We have the
\(LHS\)
\[= \text{var } u, v \cdot ((\eta_u; (P_2 \parallel (Q; Q))) \parallel (tg; (P \parallel Q))) \quad \{(\text{par-6})\}\]
\[= \text{RHS}\]

(2.2) \(Q_1 = S_1 \cap S_2\).

\(LHS\)
\[= \text{var } u, v \cdot (P || ((S_1; Q) \cap (S_2; Q))) \quad \{(\text{par-3})\}\]
\[= \text{var } u, v \cdot ((S_1; (P \parallel Q)) \cap (S_2; (P \parallel Q))) \quad \{(\text{seg-3})\}\]
\[= \text{RHS}\]

(2.3) \(Q_1 = \{S_1 \cup b \triangleright S_2\}\).

\(LHS\)
\[= \text{var } u, v \cdot (P || \bigcup_{n \geq 0} (F^n(\bot); S_2; Q))) \quad \{(\text{continuity of } \parallel \})\]
\[= \text{var } u, v \cdot (\bigcup_{n \geq 0} (P || (F^n(\bot); S_2; Q))) \quad \{(\text{continuity of } \parallel \} \text{ and } \})\]
\[= \text{RHS}\]

(2.4) \(Q_1 = b * S\).

Let
\(F(X) =_{df} (S; X) \cup b \triangleright \text{skip}\).

For \(n \geq 0\) we define
\(F^0(\bot) =_{df} \bot\),
\(F^{n+1}(\bot) =_{df} F(F^n(\bot))\).

Then
\[LHS\]
\[= \text{var } u, v \cdot (P || \bigcup_{n \geq 0} (F^n(\bot); Q))) \quad \{(\text{continuity of } \parallel \} \text{ and } \})\]
\[= \text{var } u, v \cdot (\bigcup_{n \geq 0} (P || (F^n(\bot); Q))) \quad \{(\text{continuity of } \parallel \} \text{ and } \})\]
\[= \text{RHS}\]

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

(2.6.1) \(S_1\) is one of the following forms: \(S_{10} \cap S_11, S_{10} \cup b \triangleright S_11, (g; S_{10}) || (g2; S_{11})\). By laws (seg-3) (seg-5), and (seg-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 \(\text{stop}\). Let us consider \(S_1 = tg\).

\(LHS\)
\[= \text{var } u, v \cdot (tg; (P || (S_2; Q))) \quad \{|hypothesis\}\]
\[= \text{RHS}\]

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

Similar to (2.5), we have

(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) \(P_1 = S_1 \cap S_2\).

(3.3) \(P_1 = S_1 \cup b \triangleright S_2\).

We first have
\[LHS\]
\[= \text{var } u, v \cdot (\bigcup_{n \geq 0} (P || (F^n(\bot); S_2; Q))) \quad \{|hypothesis\}\]
\[= \text{var } u, v \cdot (\bigcup_{n \geq 0} (P || (F^n(\bot); S_2; Q))) \quad \{|hypothesis\}\]
\[= \text{RHS}\]

(3.4) \(P_1 = l_{i \in I} (g_i S_i)\).

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

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

Let
\(F(X) =_{df} (S; X) \cup b \triangleright \text{skip}\).

For \(n \geq 0\) we define
\(F^0(\bot) =_{df} \bot\),
\(F^{n+1}(\bot) =_{df} F(F^n(\bot))\).

Then
\( \text{LHS} \)
\[ = \text{var}_{u,v} \cdot (\bigcup_{n \geq 0} (F^n(\perp); P)) \parallel (Q_1; Q) \]
\{continuity of \}
\[ = \text{var}_{u,v} \cdot (\bigcup_{n \geq 0} ((F^n(\perp); P)) \parallel (Q_1; Q)) \]
\{continuity of \}
\[ = \text{var}_{u,v} \cdot (\bigcup_{n \geq 0} (((F^n(\perp)); Q_1); (P \parallel Q))) \]
\{continuity of \ and \}
\[ = \text{RHS} \]

(3.6) \( P_1 = S_1 \parallel S_2 \).

(3.6.1) \( S_1 \) is one of the following forms: \( S_{10} \parallel S_{11} \), \( S_{10} \parallel b \triangleright S_{11} \), \( (g_1 S_{10}) \parallel (g_2 S_{11}) \). By laws \( \text{seq-3} \), \( \text{seq-5} \), and \( \text{seq-4} \), 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 \( \perp \) or \( \text{stop} \). 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) \( S_1 = b \ast S \).

Similar to (3.5), we have
\[ \text{LHS} \]
\[ = \text{var}_{u,v} \cdot (\bigcup_{n \geq 0} (F^n(\perp); S_2; P)) \parallel (Q_1; Q) \]
\{continuity of \}
\[ = \text{var}_{u,v} \cdot (\bigcup_{n \geq 0} ((F^n(\perp); S_2; P)) \parallel (Q_1; Q)) \]
\{continuity of \}
\[ = \text{var}_{u,v} \cdot (\bigcup_{n \geq 0} (((F^n(\perp)); S_2); (Q_1); (P \parallel Q))) \]
\{continuity of \ and \}
\[ = \text{RHS} \]

Proof of the Rule for Hardware Augmentation

Proof For simplicity, let us denote \( \text{int}(D, D') \) as \( \bar{D} \). We need to prove
\[ (C; \rightarrow \eta_e) \parallel D = (C; \rightarrow \eta_e) \parallel \bar{D} \]

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.

\[ C = \text{stop} \text{ or } C = \perp, \text{ trivial.} \]

\[ C = tg, \text{ where } tg \text{ is one of the following forms } @x := e, \rightarrow \eta_e, \#n. \text{ We have} \]
\[ \text{LHS} \]
\[ = tg; (\rightarrow \eta_e \parallel D) \]
\{ \text{(par-6), (guard-4)} \}
\[ = tg; \text{skip;} \]
\{ \text{(ivar-4)} \}
\[ = tg; (\rightarrow \eta_e \parallel \bar{D}) \]
\{ \text{(par-6), (guard-4)} \}
\[ = \text{RHS} \]

(1.2) \( C = S_1 \parallel S_2 \).

(1.3) \( C = S_1 \triangleright b \triangleright S_2 \).

\[ \text{LHS} \]
\[ = ((S_1; \rightarrow \eta_e) \parallel (S_2; \rightarrow \eta_e)) \parallel D \]
\{ \text{seq-3} \}
\[ = ((S_1; \rightarrow \eta_e) \parallel D) \parallel (S_2; \rightarrow \eta_e) \parallel D \]
\{ \text{par-10} \}
\[ = ((S_1; \rightarrow \eta_e) \parallel \bar{D}) \parallel (S_2; \rightarrow \eta_e) \parallel \bar{D} \]
\{ \text{hypothesis} \}
\[ = \text{RHS} \]

(1.4) \( C = \bigcup_{i \in I} (g_i S_i) \).

\[ \text{LHS} \]
\[ = (\bigcup_{i \in I} (g_i (S_i; \rightarrow \eta_e))) \parallel D \]
\{ \text{par-6}, \text{ (guard-4)} \}
\[ = \bigcup_{i \in I} (g_i ((S_i; \rightarrow \eta_e)) \parallel D) \]
\{ \text{hypoisthesis} \}
\[ = \bigcup_{i \in I} (g_i ((S_i; \rightarrow \eta_e)) \parallel \bar{D}) \]
\{ \text{par-6} \}

(1.5) \( C = b \ast S \).

Let
\[ F(X) = q; (S; X) \triangleright b \triangleright \text{skip}, \]

For \( n \geq 0 \), we define
\[ F^n(\perp) = q; \perp, \]
\[ F^{n+1}(\perp) = q; F(F^n(\perp)). \]

Then
\[ \text{LHS} \]
\[ = \bigcup_{n \geq 0} (F^n(\perp); \rightarrow \eta_e)) \parallel D \]
\{ \text{continuity of} \}
\[ = \bigcup_{n \geq 0} ((F^n(\perp); \rightarrow \eta_e)) \parallel D \]
\{ \text{continuity of} \}
\[ = \bigcup_{n \geq 0} (((F^n(\perp); \rightarrow \eta_e)) \parallel \bar{D}) \]
\{ \text{continuity of} \}
\[ = \text{RHS} \]

(1.6) \( C = S_1 \parallel S_2 \).

(1.6.1) \( S_1 \) is one of the following forms \( S_{10} \parallel S_{11}, S_{10} \parallel b \triangleright S_{11} \), \( (g_1 S_{10}) \parallel (g_2 S_{11}) \). By laws \( \text{seq-3}, \text{ seq-5}, \text{ and seq-4} \) 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 \( \perp \) or \( \text{stop} \). The case \( S_1 = tg \) is dealt with in what follows.

\[ \text{LHS} \]
\[ = tg; (S_2 \parallel D) \]
\{ \text{par-6}, \text{ (guard-4)} \}
\[ = tg; (S_2 \parallel \bar{D}) \]
\{ \text{hypoisthesis} \}
\[ = \text{RHS} \]

(1.6.3) \( S_1 = b \ast S \).

Similar to (1.5) we have
\begin{align*}
\text{LHS} &= (\bigcup_{n \geq 0} (F^n(\bot); S_2; \rightarrow \eta_c) \parallel D) \quad \{\text{continuity of } \parallel\} \\
&= (\bigcup_{n \geq 0} ((F^n(\bot); S_2 \rightarrow \eta_c) \parallel D) \quad \{\text{hypothesis}\} \\
&= (\bigcup_{n \geq 0} (((F^n(\bot); S_2; \rightarrow \eta_c) \parallel \hat{D})) \quad \{\text{continuity of } \parallel\} \\
&= \text{RHS}
\end{align*}

\textbf{case 2} \quad C = \rightarrow \eta_c; C_0; \eta_k.

\begin{align*}
\text{LHS} &= (\text{\textit{lvar-f}}, \text{ event control insensitive}) \\
&= (C_0; \eta_k; \rightarrow \eta_c) \parallel (M; \rightarrow \eta_k; D) \quad \{\text{Lemma 1}\} \\
&= (C_0 \parallel M; ((\eta_k; \rightarrow \eta_c) \parallel (\rightarrow \eta_k; D)) \quad \{(\text{\textit{lvar-f}}, \text{ event control insensitive}\} \\
&= (C_0 \parallel M; \text{skip}; (\rightarrow \eta_c \parallel D) \quad \{(\text{\textit{lvar-f}})\} \\
&= (C_0 \parallel M; \text{skip}; (\rightarrow \eta_c \parallel D) \quad \{(\text{\textit{lvar-f}})\} \\
&= (C_0; \eta_k; \rightarrow \eta_c) \parallel (M; \rightarrow \eta_k; \hat{D})) \quad \{\text{Lemma 1}\} \\
&= (((C_0; \eta_k; \rightarrow \eta_c) \parallel (M; \rightarrow \eta_k; \hat{D})) \quad \{(\text{\textit{lvar-f}}, \text{ event control insensitive}\} \\
&= \text{RHS}
\end{align*}

\textbf{case 3} \quad C \text{ is one of the following forms.}

\begin{enumerate}
\item[(3.1)] \quad C = C_0; C^1, \text{ analogous to that in (1.6).}
\item[(3.2)] \quad C = C_0 \cap C^1, \text{ analogous to that in (1.2).}
\item[(3.3)] \quad C = C_0 \triangleleft b \triangleright C^1, \text{ analogous to that in (1.3).}
\item[(3.4)] \quad C = \bigcup_{i \leq 1} (g_i C^i), \text{ analogous to that in (1.4).}
\item[(3.5)] \quad C = b * C, \text{ analogous to that in (1.5).} \quad \Box
\end{enumerate}