Towards An Automated Approach to Hardware/Software Decomposition

Shengchao Qin[1,2]
[1]Singapore-MIT Alliance
[2]Peking University

Jifeng He*
International Institute for Software Technology
United Nations University

Wei-Ngan Chin[1,3]
[1]Singapore-MIT Alliance
[3]National University of Singapore

Abstract

We propose in this paper an algebraic approach to hardware/software partitioning in Verilog Hardware Description Language (HDL). We explore a collection of algebraic laws for Verilog programs, from which we design a set of syntax-based algebraic rules to conduct hardware/software partitioning. The co-specification language and the target hardware and software description languages are specific subsets of Verilog. Through this, we confirm successful verification for the correctness of the partitioning process by an algebra of Verilog. Facilitated by Verilog's rich features, we have also successfully studied hw/sw partitioning for environment-driven systems.

Keywords. *Verilog, algebraic laws, hardware/software co-design, 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 behaviours of the process evolving within its environment, and an analysis of the requirement for its safety performance. From these is derived a specification of the electronic or program-centred components of the system. The project 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 is executed at a high speed by electronic circuity. In order to achieve the 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. Selecting individual gates and registers in a circuit like selecting individual machine instruction in a program. State transition diagrams are like flowcharts. These methods may have been adequate for small circuit design when they were introduced, but they are 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 very adverse effect on profits of the enterprise concerned. A method with great potential is to develop a useful collection of proven equations and other theorems, to calculate, manipulate and transform a specification formulae to the product.

Hardware/software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of the co-design process is to partition a specification into hardware and software. This paper proposes a partitioning method whose correctness is verified using algebraic laws developed for the Verilog hardware description language. One of advantages of this approach lies in that it ensures the correctness of the partitioning process. Moreover, it optimises the underlying target architecture, and facilitates the reuse of hardware devices.

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 [6] 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 [5]. Sampaio showed how to reduce the compiler design task to program transformation [15]. 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 [11], and He Jifeng *et al* provided a formal verification of the hardware compilation scheme within the algebra of Occam programs [4].

Recently, some works have suggested the use of formal methods for the partitioning process [16, 13]. In [16], 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

*On leave from Software Engineering Institute of East China Normal University.

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 [13], Qin *et al* propose an algebraic approach to partition a specification into hardware and software in one step and as well verify the correctness of the partition process. However, their 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.

The remainder of this paper is organised as follows. Section 2 introduces Verilog HDL and explores some useful algebraic laws. Section 3 describes our partitioning strategy. The co-specification language and target hardware and software architectures are proposed in section 4. Afterwards, we investigate our partition process in detail in section 5 by designing a collection of proved syntax-based partitioning rules. A simple conclusion is followed in section 6.

2 Verilog and Its Algebraic Laws

Modern hardware design typically uses a hardware description language (HDL) to express designs at various levels of abstraction. A HDL is a high level programming language with usual programming constructs, such as assignments, conditionals and iterations, and appropriate extensions for real-time, concurrency and data structures suitable for modelling hardware.

Verilog is a HDL that has been standardized and widely used in industry ([9]). 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 parallel program made up of a set of sequential processes, with or without local variable declaration.

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

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

$$\begin{array}{rl} S ::= & PC \text{(primitive command)} \\ \mid & S; S \text{ (sequential composition)} \\ \mid & \mathit{if}\ b\ S\ else\ S \text{ (conditional)} \\ \mid & \mathit{while}\ b\ S \text{ (iteration)} \\ \mid & (g\ S) [\!] \ldots [\!] (g\ S) \text{ (guarded choice)} \\ \mid & always\ S \text{ (infinite loop)} \\ \mid & \mathit{case}\ (e)\ (pt\ S) \ldots (pt\ S) \text{ (switch statement)} \end{array}$$

where

$$\begin{array}{rl} PC ::= & skip \mid chaos \mid \rightarrow \eta_v \text{ (output event)}^1 \\ & \mid v := e \text{ (instantaneous assignment)} \\ & \mid v := cg\ e \text{ (assignment with timing control)} \end{array}$$

$$g ::= \rightarrow \eta_v \mid cg \text{ (timing control)}$$

$$cg ::= \#\Delta \text{ (time delay)} \mid eg \text{ (event control)}$$

$$eg ::= @(\eta_v) \mid eg\ or\ eg \mid eg\ and\ eg \mid eg\ and\ \neg eg$$

$$\begin{array}{rl} \eta_v ::= & \sim v \text{ (value change)} \\ & \mid \uparrow v \text{ (value rising)} \\ & \mid \downarrow v \text{ (value falling)} \end{array}$$

Remark: $chaos$ is the worst program with the most unpredictable behavior. We will see that, in the algebra of Verilog programs, it is a zero element for some operators.

To facilitate algebraic reasoning, the language is enriched with

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

where the process after a guard g can be a parallel process.

Although it is reported that Verilog has been much more widely used in industry than VHDL ([1]), the formal semantics of Verilog has not been fully studied. Gordon tries to relate event semantics of Verilog to its trace semantics ([2]). He and Zhu ([7, 19]) explore an operational and a denotational semantics for Verilog and investigate some algebraic laws from them. Zhu, Bowen and He ([17, 18]) establish formal consistency between above-mentioned two presentations. Iyoda and He ([10]) successfully apply simple algebraic laws of Verilog to hardware synthesis process. Recently, He has explored a collection of algebraic laws for Verilog, by which a well-formed Verilog program can be transformed into head normal forms ([3]). In the following, we investigate some algebraic laws for Verilog, which will play a fundamental role in our hardware/software partitioning process.

Before presenting algebraic laws, we define a triggering predicate as follows.

Definition 2.1 *Given an event control eg, we define those simple events that enable eg as follows.*

$$E(eg) =_{df} \left\{ \begin{array}{l} \{\uparrow x\},\ \mathit{if}\ eg = @(\uparrow x) \\ \{\downarrow x\},\ \mathit{if}\ eg = @(\downarrow x) \\ \{\uparrow x, \downarrow x\},\ \mathit{if}\ eg = @(\sim x) \\ E(eg_1) \cup E(eg_2),\ \mathit{if}\ eg = eg_1\ or\ eg_2 \\ E(eg_1) \cap E(eg_2),\ \mathit{if}\ eg = eg_1\ and\ eg_2 \\ E(eg_1) \setminus E(eg_2),\ \mathit{if}\ eg = eg_1\ and\ \neg eg_2 \end{array} \right\}$$

[1] In order to avoid any unexpected loss of signals, we claim that an abstract output event only takes place at the moment when there're no active events at all. We will mention it again in a later section.

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

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

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

$$\eta \not\rightsquigarrow eg \ =_{df} \ E(@(\eta)) \cap E(eg) = \emptyset \qquad \square$$

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

Definition 2.2 *A guarded choice $[\!]_{i\in I}\, g_i\, P_i$ is well-formed iff all its input guards are disjoint, i.e., for any input guards g_k, g_l from $\{g_i \mid i \in I\}$, if $E(g_k) \cap E(g_l) \neq \emptyset$, 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.
(*assgn-1*) $v := e;\ v := f \ = \ v := f[e/v]$

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

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

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

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

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

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

(*seq-4*) $(\textit{if}\ b\ P\ else\ Q); R \ = \ \textit{if}\ b\ (P; R)\ else\ (Q; R)$

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

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

(1). $\rightarrow \eta; S \ = \ \begin{cases} \rightarrow \eta\, S & \text{if } \eta \not\rightsquigarrow g; \\ \rightarrow \eta\, P_k & \text{if } \eta \rightsquigarrow g_k \text{ for some } k \in I. \end{cases}$

(2). $(x < f)_\perp; @(x := f); S \ =$
$\begin{cases} (x < f)_\perp; @(x := f)\, S, \text{ if } \uparrow x \not\rightsquigarrow g; \\ (x < f)_\perp; @(x := f)\, P_k, \text{ if } \uparrow x \rightsquigarrow g_k \text{ for some } k \in I. \end{cases}$

(3). $(x > f)_\perp; @(x := f); S \ =$
$\begin{cases} (x > f)_\perp; @(x := f)\, S, \text{ if } \downarrow x \not\rightsquigarrow g; \\ (x > f)_\perp; @(x := f)\, P_k, \text{ if } \downarrow x \rightsquigarrow g_k \text{ for some } k \in I. \end{cases}$

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

where $b_\perp$ is an assertion defined as $\textit{if}\ b\ skip\ else\ chaos$ ([8]).

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

Assignment distributes forward over conditional.
(*cond-1*) $v := e; (\textit{if}\ b(v)\ P\ else\ Q) \ =$
$\quad \textit{if}\ b(e)\ (v := e; P)\ else\ (v := e; Q)$

Iteration is subject to the fixed point theorem.
(*iter-1*) $\textit{while}\ b\ P \ = \ \textit{if}\ b\ (P; \textit{while}\ b\ P)\ else\ skip$

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

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

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

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

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

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

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

(*lvar-2*) Provided x is not free in b, then
$var\ x \bullet (\textit{if}\ b\ P\ else\ Q) = \textit{if}\ b\ (var\ x \bullet P)\ else\ (var\ x \bullet Q)$

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

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

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

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

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

$$G_1 \ = \ [\!]_{i\in I}(g_i\, Q_i) \qquad G_2 \ = \ [\!]_{j\in J}(h_j\, R_j)$$
$$G_3 \ = \ [\!]_{k\in K}(e_{v_k}\, P_k) \qquad G_4 \ = \ [\!]_{l\in L}(e_{u_l}\, T_l)$$

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

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

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

(*par-6*) Let $g =_{df} or_{i\in I}\, g_i$, $h =_{df} or_{j\in J}\, h_j$, then
$(G_1 \,[\!]\, G_3) \parallel (G_2 \,[\!]\, G_4) =$
$[\!]_{i\in I}\,((g_i\ and \neg h)\,(Q_i \parallel (G_2 \,[\!]\, G_4))\,[\!]$
$[\!]_{j\in J}\,((h_j\ and \neg g)\,((G_1 [\!] G_3) \parallel R_j))\,[\!]$
$[\!]_{i\in I, j\in J}\,((g_i\ and\ h_j)\,(Q_i \parallel R_j))\,[\!]$
$[\!]_{k\in K, j\in J, e_{v_k} \rightsquigarrow h_j}\,(e_{v_k}\,(P_k \parallel R_j))\,[\!]$
$[\!]_{k\in K, e_{v_k} \nrightarrow h}\,(e_{v_k}\,(P_k \parallel (G_2 \,[\!]\, G_4)))\,[\!]$
$[\!]_{i\in I, l\in L, e_{u_l} \rightsquigarrow g_i}\,(e_{u_l}\,(Q_i \parallel T_l))\,[\!]$
$[\!]_{l\in L, e_{u_l} \nrightarrow g}\,(e_{u_l}\,((G_1 \,[\!]\, G_3) \parallel T_l))$

(*par-7*) An assignment thread is involved.
(1) $(x := e) \parallel (y := f) =$
$(@(x := e)\,(y := f)) \,[\!]\, (@(y := f)\,(x := e))$
(2) $(x := e) \parallel G_2 =$
$(@(x := e)\,G_2) \,[\!]\, [\!]_{j\in J}\,(h_j\,((x := e) \parallel R_j))$

The parallel operator is distributive over non-deterministic choice.
(*par-8*) $(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$

In some special case, the parallel operator distributes over conditional.
(*par-9*) $var\ v_1, \ldots, v_n \bullet ((if\ b\ S_1\ else\ S_2) \parallel G) =$
$var\ v_1, \ldots, v_n \bullet (if\ b\ (S_1 \parallel G)\ else\ (S_2 \parallel G))$,
provided guards in G are either event controls with respect to variables in $\{v_1, \ldots, v_n\}$ or time-delay guards.

Time-delay guards are involved in the following law.
(*par-10*) Let $\Delta_1 > \Delta_2 > 0, \Delta > 0$.
(1). $(\#\Delta\ S)[\!]G_3 = G_3$
(2). $(G_1 \,[\!]\, \#\Delta_1\ S) \parallel (G_2 \,[\!]\, \#\Delta_2\ T) =$
$[\!]_{i\in I}\,((g_i\ and \neg h)\,(Q_i \parallel (G_2 \,[\!]\, \#\Delta_2\ T))\,[\!]$
$[\!]_{j\in J}\,((h_j\ and \neg g)\,((G_1 \,[\!]\, \#\Delta_1\ S) \parallel R_j))\,[\!]$
$[\!]_{i\in I, j\in J}\,((g_i\ and\ h_j)\,(Q_i \parallel R_j))\,[\!]$
$[\!]\ \#\Delta_2\,((\#(\Delta_1 - \Delta_2)\ S) \parallel T)$
(3). $(G_1 \,[\!]\, \#\Delta\ S) \parallel (G_2 \,[\!]\, \#\Delta\ T) =$
$[\!]_{i\in I}\,((g_i\ and \neg h)\,(Q_i \parallel (G_2 \,[\!]\, \#\Delta\ T))\,[\!]$
$[\!]_{j\in J}\,((h_j\ and \neg g)\,((G_1 \,[\!]\, \#\Delta\ S) \parallel R_j))\,[\!]$
$[\!]_{i\in I, j\in J}\,((g_i\ and\ h_j)\,(Q_i \parallel R_j))\,[\!]$
$[\!]\ \#\Delta\,(S \parallel T)$

The guarded choice is idempotent, symmetric and associative.
(*guard-1*) $G_1 \,[\!]\, G_1 = G_1$
(*guard-2*) $G_1 \,[\!]\, G_2 = G_2 \,[\!]\, G_1$
(*guard-3*) $(g_1\ Q_1) \,[\!]\, ((g_2\ Q_2) \,[\!]\, (g_3\ Q_3)) =$
$((g_1\ Q_1) \,[\!]\, (g_2\ Q_2)) \,[\!]\, (g_3\ Q_3)$
(*guard-4*) $var\ v \bullet ((@(\eta_v)\ P) \,[\!]\, G_1) = var\ v \bullet G_1$

The construct $always\ S$ executes S forever.
(*always-1*) $always\ S = S; always\ S$

From the operational semantics of Verilog ([7]), we know the fact that $skip$ is not a left zero of sequential composition in general cases, because it might filter some signal. Hereby, the following inequation is obvious.

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

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

Definition 2.3 (Event control insensitive) *A process P is event control insensitive if $skip; P = P$.* □

Theorem 2.4 *The following processes are event control insensitive.*

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

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

Lemma 2.5 *Let* $P = (@\eta_u\ P_2)$, $Q = (\rightarrow\eta_u; @\eta_v\ Q_2)$, *suppose sequential programs* P_2, Q_1 *are event control insensitive,* P_1 *is a sequential process not containing any timing controls, and variables* u, v *do not occur in* P_1 *or* Q_1, *then*

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

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

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

Proof: The proof is presented in [12].

We introduce an ordering relation between programs before further investigation.

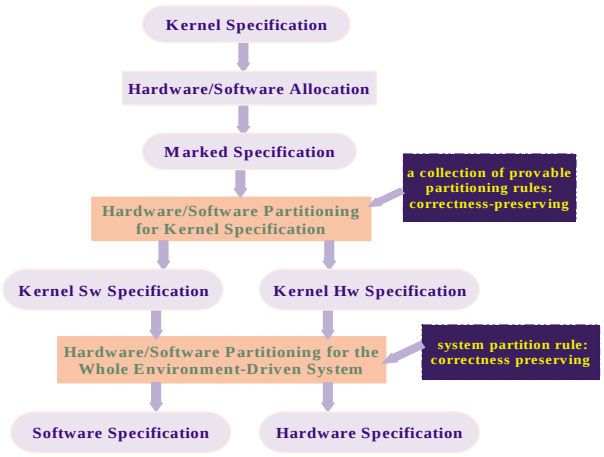

Figure 1. Hw/Sw Partitioning Strategy

Definition 2.6 (Refinement) *Let P, Q be Verilog processes employing the same set of variables, we say Q is a refinement of P, denoted as $P \sqsubseteq Q$, if $P \sqcap Q = P$ is algebraically provable.* □

3 Partitioning Strategy

This section is devoted to introduce our hardware/software partitioning strategy, which can be described in four steps, see Fig. 1.

- Before conducting the partitioning process, the programmer codes the kernel specification for the system in our co-specification language, which is a sequential subset of Verilog and will be explained in detail in next section.
- Then, assisted by program analysis techniques ([13]), 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 successfully propose an algebraic approach to hardware/software partitioning, which ensures the correctness of the hardware/software partitioning process and facilitates the automatic partitioning.

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

4 The Decomposition Framework

In this section, we intend to introduce our hardware/software partitioning framework. We propose our co-specification language and investigate the underlying target hardware/software architectures.

The co-specification language we adopt is a sequential subset of Verilog, which comprises the following syntactic elements.

$$\begin{aligned} S ::= \;& AC \text{ (primitive command)} \\ \mid\; & S; S \text{ (sequential composition)} \\ \mid\; & \mathit{if}\, b\, S\, \mathit{else}\, S \text{ (conditional)} \\ \mid\; & S \sqcap S \text{ (non-deterministic choice)} \\ \mid\; & \mathit{while}\, b\, S \text{ (iteration)} \\ \mid\; & (g\, S) \,[\!]\, (g\, S) \text{ (guarded choice)} \end{aligned}$$

where

$$\begin{aligned} AC ::= \;& v := e \mid \to \eta_v \mid @\eta_v \mid \#\Delta \mid \mathit{chaos} \\ \mid\; & (v := e)_n \text{ (assign. with timing constraint)} \\ \mid\; & \langle S \rangle \text{ (specific block)} \end{aligned}$$

$$\eta_v ::= \sim v \mid \uparrow v \mid \downarrow v$$

The assignment statement with time constraint $(v := e)_n$ doesn't appear explicitly in Verilog's syntax introduced in section 2, but it is in fact a well-formed Verilog program since

$$(v := e)_n = \sqcap_{0 \le k \le n} (v := \#k\, e)$$

Moreover, the block notation in $\langle S \rangle$ has no semantical meanings.

From the customer's requirements, the programmer can describe the kernel specification for the system to be designed in this co-specification language. After appropriate hardware/software marking and allocation, a marked source program is passed to the partitioning process.

The underlying target hardware and software components from the kernel specification will own specially-chosen forms. We adopt an event-trigger mechanism to synchronise behaviours 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). $\to \eta_r; C; @\eta_a$, where C is a member of $CP(r, a)$ not mentioning r, a, or any timing controls;

(3). $C_1; C_2$, or $\mathit{if}\, b\, C_1\, \mathit{else}\, C_2$, or $C_1 \sqcap C_2$, or $(g_1\, C_1) \,[\!]\, (g_2\, C_2)$, where $C_1, C_2, g_1, g_2 \in CP(r, a)$;

(4). *while* $b\, C$, where $C \in CP(r, a)$.

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

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

$$C_0 \parallel D_0$$

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

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

where $M =_{df}$ *case* $(id)\ (p_1\ M_1) \ldots (p_n\ M_n)$ is a case construct not containing r, a, ε. D_0 represents a digital device which offers a set of services $M_1, \ldots, M_n$ to its environment. It responds to a request by matching the current value of shared variable id (a natural number assigned by the software) with the patterns $p_1, \ldots, p_n$ to choose a corresponding method to serve.

We denote as $DP(r, a, \varepsilon)$ 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

$$\Psi^s_f(S) \;=_{df}\; always\ (@\eta_s\ S; \rightarrow \eta_f)$$

which is driven by the external environmental process:

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

and derive the partitioning of $\Psi^s_f(S)$ under the environment Env as

$$\Psi^s_f(C) \parallel_{Env} D$$

where $P \parallel_{Env} Q \;=_{df}\; P \parallel Env \parallel Q$; the software component enjoys the form

$$\Psi^s_f(C) \;=_{df}\; always\ (@\eta_s\ C; \rightarrow \eta_f)$$

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

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

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

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

Theorem 4.1 *For any* C_1, C_2 *in* $CP_\varepsilon(r, a)$ *and* D_0 *in* $DP(r, a, \varepsilon)$, *we have*

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

Proof: By structural induction on C_1. The detailed proof is presented in [14]. □

The following corollary is directly from theorem 4.1.

Corollary 4.2 *Given* $C \in CP_\varepsilon(r, a)$, $D_0 \in DP(r, a, \varepsilon)$, *we have*

$$(\textit{while}\, b\, C; \rightarrow \eta_\varepsilon) \parallel D_0 = \textit{while}\, b\, ((C; \rightarrow \eta_\varepsilon) \parallel D_0)$$ □

5 Hardware/Software Partitioning

This section specifies our hardware/software partitioning process in detail. As mentioned in section 3, the task is divided to two steps: hardware/software partitioning for kernel specification; decomposition of the whole system's specification. The process will be investigated in detail in the following two subsections.

5.1 Splitting Rules for Kernel Specification

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

We introduce a predicate $Split$ which plays a vital role in formalising the splitting rules.

Definition 5.1 ($Split$**)** *Let* $V = \{r, a, \varepsilon, id\}$. *Given a program* S *in the co-specification language, its hardware/software partition* $((C; \rightarrow \eta_\varepsilon),\ D^0)$ *is specified by the following predicate:*

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

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

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

5.1.1 The Software-Extraction Splitting Rules

The *software-extraction* 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.

Software-Extraction Rule for Sequential Composition

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

Proof

$$\begin{array}{ll} S_1\ ;\ S_2 & \{;\ \textit{is monotonic}\} \\ \sqsubseteq\ ((C_1; \rightarrow \eta_\varepsilon) \parallel D_0);\ ((C_2; \rightarrow \eta_\varepsilon) \parallel D_0) & \{theorem\ 4.1\} \\ =\ (C_1;\ C_2; \rightarrow \eta_\varepsilon) \parallel D_0 & \square \end{array}$$

Software-Extraction Rule for Conditional

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

Software-Extraction Rule for Non-Deterministic Choice

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

Software-Extraction Rule for Guarded Choice

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

Proof The proofs for the above three rules are presented in [14].

Software-Extraction Rule for Iteration

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

Proof It's straightforward from corollary 4.2. $\square$

5.1.2 The Software-Hardware-Extraction Splitting Rules

In the *software-hardware-extraction* style, both the hardware and software components of the source program are integrated from those of its constituents.

Before investigating the *software-hardware-extraction* splitting rules, we introduce the notion of *mergeable* on hardware components from $DP(r, a, \varepsilon)$.

Definition 5.2 *Let*

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

where

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

D^1 *and* D^2 *are said to be* mergeable, *denoted by* $mergeable(D^1, D^2)$, *if*

$Var(D^1) = Var(D^2)$, *and*

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

In such a case, we define

$D = \textit{integrate}(D^1, D^2) =_{df}$
$\mu X \bullet ((@\eta_r\ M; \rightarrow \eta_a; X) [\!] (@\eta_\varepsilon\ skip))$,
where $M =_{df}$ *case* $(id)\ (t_1\ M_1) \ldots (t_r\ M_r)$,
and
$\{t_1, \ldots, t_r\} = \{p_1^1, \ldots, p_{n_1}^1\} \cup \{p_1^2, \ldots, p_{n_2}^2\}$,
and
$\{M_1, \ldots, M_r\} = \{M_1^1, \ldots, M_{n_1}^1\} \cup \{M_1^2, \ldots, M_{n_2}^2\}$. $\square$

First of all, we present a basic rule for hardware augmentation, from this and the *software-extraction* rules in the former section, we can directly obtain the corresponding *software-hardware-extraction* rules in all cases.

Auxiliary Rule for Hardware Augmentation

$$\frac{Split_V(S,\ C,\ D) \quad mergeable(D, D')}{Split_V(S,\ C,\ \textit{integrate}(D, D'))}$$

Proof The proof can be reached in [12]. $\square$

SW-HW-Extraction Rule for Sequential Composition

$$\frac{Split_V(S_i, C_i, D_i) \quad Var(S_1) = Var(S_2) \quad mergeable(D_1, D_2)}{Split_V(S_1; S_2,\ C_1; C_2,\ \textit{integrate}(D_1, D_2))}$$

SW-HW-Extraction Rule for Conditional

$$\frac{Split_V(S_i,\ C_i,\ D_i) \quad Var(S_1) = Var(S_2) \quad mergeable(D_1, D_2)}{Split_V(\textit{if}\ b\ S_1\ else\ S_2,\ \textit{if}\ b\ C_1\ else\ C_2,\ \textit{integrate}(D_1, D_2))}$$

SW-HW-Extraction for Non-Deterministic Choice

$$\frac{Split_V(S_i,\ C_i,\ D_i) \quad Var(S_1) = Var(S_2) \quad mergeable(D_1, D_2)}{Split_V(S_1 \sqcap S_2,\ C_1 \sqcap C_2,\ \textit{integrate}(D_1, D_2))}$$

SW-HW-Extraction Rule for Guarded Choice

$$\frac{Split_V(S_i,\ C_i,\ D_i) \quad Var(S_1) = Var(S_2) \quad mergeable(D_1, D_2)}{Split_V((g_1\ S_1) [\!] (g_2\ S_2),\ (g_1\ C_1) [\!] (g_2\ C_2),\ \textit{integrate}(D_1, D_2))}$$

The software-hardware-extraction rule for iteration enjoys exactly the same form as the corresponding software-extraction rule.

5.1.3 Splitting Atomic Commands

The details for specific blocks' partitioning are similar to discussions in [13].

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

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

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

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

$D =_{df} \mu X \bullet$

$$\left(\begin{array}{l} (@\eta_r\ case\ (id)\ (1\ (ly := f(x,c))_n); \rightarrow\eta_a; X) \\ [\!] (@\eta_\varepsilon\ skip) \end{array} \right).$$

Case 2: f is a hardware-marked function, but x is allocated to software.

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

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

$D =_{df} \mu X \bullet$

$$\left(\begin{array}{l} (@\eta_r\ case\ (id)\ (1\ (ly := f(lx,c))_n); \rightarrow\eta_a; X) \\ [\!] (@\eta_\varepsilon\ skip) \end{array} \right).$$

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

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

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

$D =_{df} \mu X \bullet$

$$\left(\begin{array}{l} (@\eta_r\ case\ (id)\ (1\ (lx := x)_0); \rightarrow\eta_a; X) \\ [\!] (@\eta_\varepsilon\ skip) \end{array} \right).$$

5.1.4 A Small Example

Given a kernel specification for a system as follows:

```
w := u + v;
#1; →(w_ready);
@(z_ready); if (z = u) (v := u × v) else skip;
while ((b && w ≤ u × v)n1){
        u := u + w;
        w := ((u − v) × (u + v))n2;
        #1; →(w_ready)
}
```

We suppose hardware/software allocation has been tackled as below, in accordance with the results of static analysis and the programmer's decision:

- variable allocation: only one variable v is allocated to hardware, others are left to software;
- computation allocation: the complicated expressions, (b && w ≤ u × v)$_{n_1}$) and ((u − v) × (u + v))$_{n_2}$, will be evaluated by hardware, others are left to software.

By applying afore-mentioned splitting rules in either style, we obtain the following hardware and software kernel specifications.

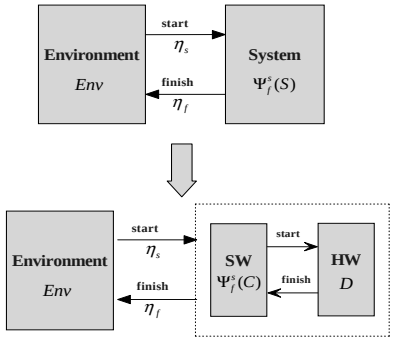

Figure 2. Hardware/Software Partition for the Whole System

Hardware Part:

```
μX•
  ((@ηr case (id)
        { 1  lv := v
          2  v := lv
          3  (lb := b && w ≤ lu × v)n1
          4  (lw := (lu − v) × (lu + v))n2 } ;
    →ηa; X)
  [](@ηε skip))
```

Software Part:

```
id := 1; →ηr; @ηa; w := u + lv;
#1; →(w_ready); @(z_ready);
if (z = u) ( id := 1; →ηr; @ηa;
             lv := u × lv;
             id := 2; →ηr; @ηa ) else skip;
id := 3; lu := u; →ηr; @ηa;
while (lb){
        u := u + w; id := 4; lu := u; →ηr; @ηa;
        w := lw; #1; →(w_ready);
        id := 3; lu := u; →ηr; @ηa
}
```

5.2 Hw/Sw Partitioning for the Whole System

Now we investigate hardware/software partitioning for the whole system. The partitioning process is illustrated in Fig. 2.

As discussed in sec. 4, suppose the whole system is specified by

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

which is driven by environment process

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

The system's behavior is specified by an infinite loop (an *always* construct). In each iteration cycle, the system responds to the start signal η_s from the external environment

by running the kernel specification S, and generating the finish signal η_f to the external environment afterwards.

For a kernel specification S, suppose we have obtained its hardware/software decomposition as follows by applying those rules in section 5.1:

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

where $V = \{r, a, \varepsilon, id\}$,
and $D = \mu X \bullet ((@\eta_r\ M; \rightarrow \eta_a; X) \ [\!] \ (@\eta_\varepsilon\ skip))$.

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

System Partitioning Rule

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

where

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

Proof

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

$$\begin{array}{l}always_{\ 0}(S) =_{df} chaos,\\ always_{\ n+1}(S) =_{df} S; always_{\ n}(S)\end{array}$$

then by law (*always-1*) ([14]), we have

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

Now by continuity of the parallel operator and law (*seq-2*) ([14]) , we only need to prove, for all $n \geq 0$,

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

where $\begin{array}{l}\Psi^s_f(P)_n =_{df} always_{\ n}(@\eta_s\ P; \rightarrow \eta_f)\\ Env_n =_{df} always_{\ n}(\rightarrow \eta_s; @\eta_f)\end{array}$

By mathematical induction on n.

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

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

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

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

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

$(\Psi^s_f(C)_{n+1}; \rightarrow \eta_\varepsilon) \parallel Env_{n+1}$
$\{(\textit{par-6}),\ (\textit{lvar-4}),\ \textit{Theorem}\ 2.4\}$
$=$ $(C; \rightarrow \eta_f; \Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel (@\eta_f; Env_n)$
$\{\textit{Lemma}\ 2.5\}$
$=$ $C; ((\rightarrow \eta_f; \Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel (@\eta_f; Env_n))$
$\{(\textit{par-6}),\ (\textit{lvar-4}),\ \textit{Theorem}\ 2.4\}$
$=$ $C; ((\Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel Env_n)$
$\{\textit{hypothesis},\ (\textit{seq-1})\}$
$=$ $always_{\ n+1}(C); \rightarrow \eta_\varepsilon$

Then, we have

$\Psi^s_f(S)_{n+1} \parallel Env_{n+1}$
$\{(\textit{par-6}),\ (\textit{lvar-4}),\ \textit{Theorem}\ 2.4\}$
$=$ $(S; \rightarrow \eta_f; \Psi^s_f(S)_n) \parallel (@\eta_f; Env_n)$
$\{\textit{Lemma}\ 2.5\}$
$=$ $S; ((\rightarrow \eta_f; \Psi^s_f(S)_n) \parallel (@\eta_f; Env_n))$
$\{(\textit{par-6}),\ (\textit{lvar-4}),\ \textit{Theorem}\ 2.4\}$
$=$ $S; (\Psi^s_f(S)_n \parallel Env_n)$
$\{\textit{precondition},\ ;\ \textit{is mono.}\}$
$\sqsubseteq$ $((C; \rightarrow \eta_\varepsilon) \parallel D); (\Psi^s_f(S)_n \parallel Env_n)$
$\{\textit{hypothesis}\}$
$\sqsubseteq$ $((C; \rightarrow \eta_\varepsilon) \parallel D); ((\Psi^s_f(C)_n; \rightarrow \eta_\varepsilon) \parallel D \parallel Env_n)$
$\{(\dagger)\}$
$=$ $((C; \rightarrow \eta_\varepsilon) \parallel D); ((always_{\ n}(C); \rightarrow \eta_\varepsilon) \parallel D)$
$\{\textit{Theorem}\ 4.1\}$
$=$ $(always_{\ n+1}(C); \rightarrow \eta_\varepsilon) \parallel D$
$\{(\dagger)\}$
$=$ $(\Psi^s_f(C)_{n+1}; \rightarrow \eta_\varepsilon) \parallel D \parallel Env_{n+1}$ □

6 Conclusion and Future Work

This paper proposes an algebraic approach to hardware/software partitioning in Verilog algebra. Verilog HDL is a hardware description language widely used by industry. Due to its rich language features, Verilog can either be used to capture system specification or adopted to specify subsequent designs of distinct levels of abstraction, including RTL design.

We adopt a sequential subset of Verilog as the specification language, and allow it to contain time constraints, so as to describe timing specification. We confine target hardware and software specifications in specially chosen subsets of Verilog, and use Verilog's event-trigger mechanism to synchronise behaviours between them. Whereas, communications between hardware and software is 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.

The partitioning process in this paper is rather different from our former approach in [13], where we only dealt with partitioning for a sequential source program. However, 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.

As part of future work, we need to consider optimization and reconfiguration of the hardware specification we generate before the process of hardware synthesis. Meanwhile, in order to apply this algebraic approach to hardware synthesis, we will have to investigate more helpful algebraic laws for Verilog. He *et al* have made noticeable progress ([3, 10]). As another emphasis in future work, we would like to involve more program analysis techniques into our co-design process, not only strengthening the existing analysis for hardware/software allocation ([13]), to obtain a more reasonable partitioning, but also introducing appropriate analyses into the co-synthesis process, to gain fine performance/cost estimation and to approach an automated design space exploration.

References

[1] M. Gordon, "The Semantic Challenge of VERILOG HDL", In *the Proc. of Tenth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 136–145, 1995.

[2] M. Gordon, "Relating Event and Trace Semantics of Hardware Description Languages", *The Computer Journal*, pp. 27–36, Vol. 45, No. 1, 2002.

[3] He J., "An Algebraic Approach to the Verilog Programming", will appear in the *Proc. of Lisbon Workshop*, 2002.

[4] He J., I. Page and J. Bowen, "A Provable Hardware Implementation of Occam", *LNCS* 711, pp. 693–703, 1993.

[5] He J. and J. Bowen, "Specification, Verification and Prototyping of an Optimised Compiler", *Formal Aspect of Computing* 6, pp. 643–658, 1994.

[6] He J. *et al*, "Provably Correct Systems", *LNCS* 863, pp. 288–335, 1994.

[7] He J. and Zhu H., "Formalising Verilog", in *the Proc. of ICECS 2000*, IEEE Computer Society Press, pp. 412–415, Lebanon, Dec. 2000.

[8] C.A.R. Hoare and He J., *Unifying Theories of Programming*, Prentice Hall, 1998.

[9] IEEE Computer Society, *IEEE Standard Hardware Description Language Based on the VERILOG Hardware Description Language (IEEE std 1364-1995)*, 1995.

[10] J. Iyoda and He J., "Towards an Algebraic Synthesis of Verilog", in *the Proc of ERSA'2001*, Las Vegas, USA, 2001.

[11] I. Page and W. Luk, "Compiling Occam into FPGAs", in *FPGAs*, eds., W. Moore and W. Luk, pp. 271–283, Abingdon EE&CS books, 1991.

[12] Qin Shengchao, "An Algebraic Approach to Hardware/Software Partitioning in Hardware/Software Co-Design", Ph.D thesis, School of Mathematical Sciences, Peking University, March, 2002.

[13] S. Qin and J. He, "Partitioning Program into Hardware and Software", in *the Proc of APSEC 2001*, IEEE Computer Society Press, pp. 309–316, Macau, 4-7 Dec., 2001.

[14] Qin S., He J., Qiu Z., and Zhang N., "Hardware/Software Partitioning in Verilog", Research Report 2002-33, School of Mathematical Sciences, http://www.math.pku.edu.cn/printdoc/182.ps. A short version is published at *LNCS* 2495, pp. 168–179, ICFEM2002, Shanghai, China.

[15] A. Sampaio, *An Algebraic Approach to Compiler Design*, World Scientific, 1997.

[16] L. Silva, A. Sampaio and E. Barros, "A Normal Form Reduction Strategy for Hardware/software Partitioning", *Formal Methods Europe (FME) 97, LNCS* 1313, pp. 624–643, 1997.

[17] Zhu H., J. Bowen and He J., "From Operational Semantics to Denotational Semantics for Verilog", in *the Proc. of CHARME 2001, LNCS* 2144, pp. 449–464.

[18] Zhu H., J. Bowen and He J., "Deriving Operational Semantics from Denotational Semantics for Verilog", in *the Proc. of APSEC 2001*, IEEE Computer Society Press, pp. 177–184, Macau, 4-7 Dec., 2001.

[19] Zhu H. and He J., "A DC-based Semantics for Verilog", in *the Proc. of the International Conference on Software: Theory and Practice (ICS2000)*, pp. 421–432, Beijing, Aug. 21-24, 2000.