Constructing Hardware/software Interface Using Protocol Converters

Shengchao Qin
Department of Informatics
School of Mathematical Sciences
Peking University
Beijing, 100871, China
qinshc@pubms.pku.edu.cn

Zongyan Qiu
Department of Informatics
School of Mathematical Sciences
Peking University
Beijing, 100871, China
zyqiu@pku.edu.cn

Jifeng He
International Institute for Software Technology
The United Nations University
P.O.Box 3058, Macau
jifeng@iist.unu.edu

Abstract

Hardware and software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of co-design process is to decompose a program into hardware and software. This paper proposes an algebraic partitioning method whose correctness is verified in the algebra of programs. We introduce the program analysis phase before program partitioning and develop a collection of syntax-based splitting rules, where the former provides the information for moving operations from software to hardware and reducing the interaction between components, and the latter supports a compositional approach to the program partitioning.

Keywords: *hardware/software co-design, hardware/software partition, program algebra.*

1. Introduction

The design of a complex software product like a nuclear reactor control system is ideally decomposed into a progression of related phases. It starts with an investigation of the properties and behaviours of the process evolving within its environment, and an analysis of requirement for its safety performance. From these is derived a specification of the electronic or program-centered 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 the chosen computer, it is executed at high speed by electronic circuitry. 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.

With chip size reaching one million transistors, the complexity of VLSI algorithms is approaching that of software algorithms. However, the design methods for circuits 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 interest in the formal verification of embedded systems is gaining ground since an error in a widely used hardware device can have significant repercussions on the stock value of the company concerned. In principle, proof of correctness of a digital device can always be achieved by making a comparison of the behavioral description of the circuit with its specification. But for a large system this would be impossibly laborious. What we need is a useful collection of proven equations and other theorems, which can be used to calculate, manipulate and transform the 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 co-design process is to partition a program into hardware and software. This paper proposes a partitioning method whose correctness is verified using the algebraic laws developed for the high level programming language. To meet performance goals, and reduce the communication between components, our approach combines the program analysis technique with the syntax-based splitting rules to move heavy-weight operations from software to hardware. The allocation of variables is also based on the data flow analysis of the source program. One of the advantages of our method is the integration of the splitting phase with the joining phase of the partitioning process. It optimizes 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 on "Provably Correct Systems". The original **ProCoS** project [**?**] 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 [**?**]. Sampaio showed how to reduce the compiler design task to one of program transformation; his formal framework is also a procedural language and its algebraic laws [**?**]. 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 Field Programmable Gate Arrays [**?**], and He Jifeng *et al* provided a formal verification of the hardware compilation scheme within the algebra of Occam programs [**?**].

Recently, some works have suggested the use of formal methods for the partitioning process [**?**, **?**, **?**]. Balboni *et al* adopt Occam as an internal model for the system exploration and partitioning strategy. Cheung pursues the structural transformation and verification within the functional programming framework. However, neither has provided a formal proof for the correctness of the partitioning process. In [**?**], Silva *et al* provide a formal strategy for carrying out the splitting phase automatically, and presents 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 to accommodate huge number of parallel processes actually increase the data flow between the hardware and software components.

The remainder of this paper is organized as follows. Section 2 describes the splitting strategy. Section 3 introduces the programming language we adopt and explores its algebraic laws. Section 4 poses the static analysis that we perform on the source program. Section 5 investigates the underlying target architecture of hardware/software components. Section 6 provides the syntax-based hardware/software splitting rules in both bottom-up and top-down styles.

2. Original Framework for Hardware/software Partitioning

This section is a brief introduction to our approach on hardware/software partitioning proposed in [**?**], which provides shades of results that can be reused, and can be regarded as a starting point of this work.

The hardware/software partitioning approach is pursued within a parallel communicating language, denoted by $\mathcal{L}$, which contains the following syntactic elements:

1. Sequential Process:

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

where $\mathit{PC} \; ::= (x := e) \mid \mathit{skip} \mid \mathit{chaos} \mid c\,!\,e \mid c\,?\,x$ and g is skip or a communication event $c\,!\,e$ or $d\,?\,x$.

2. Parallel Program:

$$P \; ::= \; S \mid P \parallel P \mid \mathbf{var}\; x \bullet P$$

In later discussions, we adopt $\mathit{Var}(P)$ and $\mathit{Chan}(P)$ to denote the set of variables and channels employed by P, respectively.

Our former hardware/software partitioning process is illustrated in Figure 1. The specification to be partitioned is a sequential program within the language $\mathcal{L}$. A collection of syntax-based partitioning rules has been developed to partition the specification into hardware and software components.

It is required that both the hardware and software components satisfy the handshaking protocol [**?**, **?**] and enjoy specially-chosen forms of our language $\mathcal{L}$.

The software component is chosen from the sequential subset of $\mathcal{L}$. It is a member of the set of processes $CP(B)$, for a given set of channels $B = \{cr_j, ca_j \mid j \in I\}$, where the set $\mathit{CP}(B)$ is the minimal set generated by following rules:

(1). a communicating process C, which does not use any channel in B, but $\mathit{Chan}(C) \supseteq B$.

(2). $cr_j\,!\,e;\; C;\; ca_j\,?\,x$, where C is a member of $\mathit{CP}(B)$ not interacting via any channel in B.

(3). $C_1; C_2$, or $C_1 \sqcap C_2$, or $\mathit{if}\; b\, C_1\, \mathit{else}\, C_2$, or $(g_1\ C_1)[\!](g_2\ C_2)$, where both g_i and C_i lie in $\mathit{CP}(B)$, for $i = 1, 2$.

(4). $\mathit{while}\; b\, C$, where C is a member of $\mathit{CP}(B)$.

The hardware component is chosen from the set of processes $H_V(B')$, for a given set of channels $B' = \{cr'_j, ca'_j \mid j \in I\}$, and the set of data variables $V = \{x_j, y_j \mid j \in I\}$. The set $H_V(B')$ is composed by the processes of the form:

Figure 1. The original hardware/software partitioning process

$$D_V^{B'} \;=\; \mu X \bullet ([\!]_{j\in I}(cr'_j?x_j; M_j; ca'_j!y_j; X)[\!]skip)$$

where none of M_j mentions channels in B'. The communicating process $D_V^{B'}$ represents a digital device which offers a set of services, each of which responds to a request from its environment on an input channel cr'_j by running the corresponding program M_j and delivering the result to the output channel ca'_j afterwards.

To simplify the interface design, in [**?**], we confined the interactions between the hardware and software components to the communications along the channels from the set B. However, in general, the hardware component is chosen from $H_V(B')$, which employs a different set of channels B'. Thus, an explicit interface component is needed to deal with the interaction between hardware and software. Such an interface is of the form

$$c\text{–}c(B, B') \;=_{df}$$
$$\mu X \bullet ([\!]_{i\in I}(cr_i\,?\,x_i; cr'_i\,!\,x_i; ca'_i\,?\,y_i; ca_i\,!\,y_i; X)\;[\!]\;skip)$$

which passes the request along a channel cr_i into the corresponding channel cr'_i, then delivers the acknowledgement from a channel ca'_i back into ca_i. The notion *c–c* means "from channel-based to channel-based".

The syntax-based hardware/software partitioning rules were developed in two different styles: the bottom-up and top-down one. We illustrate one for each as follows.

Bottom-up Rule for Sequential Composition

$$\frac{\begin{array}{c} \mathit{Split}_B(S_i,\ C_i,\ D),\ i=1,2 \\ \mathit{Var}(S_1)=\mathit{Var}(S_2),\ \mathit{Chan}(C_1)=\mathit{Chan}(C_2) \end{array}}{\mathit{Split}_B(S_1;S_2,\ C_1;C_2,\ D)}$$

Top-down Rule for Conditional

$$\frac{\begin{array}{c} \mathit{Split}_{B_i}(S_i,\ C_i,\ D_i),\ i=1,2 \\ \mathit{Var}(S_1)=\mathit{Var}(S_2),\ \mathit{Chan}(S_1)=\mathit{Chan}(S_2) \\ \mathit{consist}(D_1,D_2),\ \mathit{Var}(b)\subseteq \mathit{Var}(C_1) \end{array}}{\mathit{Split}_{B_1\cup B_2}(\mathit{if}\ b\,S_1\ \mathit{else}\ S_2,\ \mathit{if}\ b\,C_1\ \mathit{else}\ C_2,\ union(D_1,D_2))}$$

where the predicate $\mathit{Split}_B(S,\ C,\ D)$ is defined in [**?**], which specifies that the process with C and D in parallel is a refinement of the specification S, the predicate $\mathit{consist}(D_1, D_2)$ indicates that D_1 and D_2 employ the same set of variables and the same set of external channels, i.e., the channels not used for communications with the software component. $union(D_1, D_2)$ represents a digital device providing the union set of services supplied by both D_1 and D_2.

The complete set of rules can be reached in [**?**]. Based on those rules and the new interface we will construct, we can generate the new partitioning rules within a hybrid parallel language framework, where the software component is written in a high level communicating language as above, however, the hardware component is coded within a hardware description language (HDL) as Verilog, which will facilitate the hardware synthesis.

3. The Hybrid Partitioning Architecture

This section presents our hybrid partitioning framework, in which we will partition a specification coded in the sequential subset of $\mathcal{L}$ into three components: a software component owning the same form as that in our former work, a hardware component written in the Verilog HDL, and an interface component to tackle the interactions between hardware and software.

It is worth noting that the hardware and software component will adopt different communicating mechanisms, the signal-based communicating mechanism and the channel-based one respectively. Due to this fact, the partitioning framework is called hybrid([**?**]), and the interface component proves to be critical in the sense that it transforms information between two processes under different protocols.

For ease of understanding, we give the source language and the target ones separately, instead of the whole framework. Our source language is the sequential subset of the language $\mathcal{L}$ defined in last section.

The software component generated by our partitioning process will own the same form as that in the original framework, i.e, will be a member of the set of communicating processes $CP(B)$, for the given channel set B as defined earlier, which communicates with the environment via communications on channels.

However, the hardware component delivered by our partitioning process will enjoy a quite different form from that in the approach described in last section. It will be a specially chosen process from Verilog HDL, rather than a process in $\mathcal{L}$.

Given a set of signal variables $E = \{er_i, ea_i \mid i \in I\} \cup \{ter\}$, and a set of data variables $V = \{x_i, y_i \mid i \in I\}$, we define a set of processes $\mathcal{D}(E, V)$, which is composed by processes of the following form:

$$D_V^E =_{df}$$
$$\mu X \bullet (\|_{i\in I}(@er_i; M_i(x_i, y_i); \rightarrow ea_i; X) \| (@ter; skip))$$

where $@er_i$ and $\rightarrow ea_i$ are respectively an input signal and an output signal in Verilog. The signal ter is used for synchronized termination. Each $M_i(x_i, y_i)$ has input parameter x_i and output parameter y_i, and is a member of the set of Verilog processes $MP(V)$, which is constructed by the following rules:

(1). $v := e$, or *skip*, or *chaos*, where no signal variables from E occur in v or e;

(2). $M; M'$, or *if* $b\, M$ *else* M', or *while* $b\, M$, where $M, M' \in MP(V)$.

The signal-driven process D_V^E represents a digital device which offers a set of services to its environment, each of which responds to a request from its environment via an input signal er_j by running the corresponding program M_j, then putting the result to variable y_j and delivering an acknowledgement through the output signal ea_j afterwards.

Since the hardware and software components adopt different communication mechanisms, the interface component to tackle their interactions should enjoy a hybrid form and convert information between these two levels.

4. The Interface Component as a Protocol Converter

This section pursues the construction of the interface component. From the hybrid architecture mentioned above, we know that the interface component plays an essential role between the hardware and software as a protocol converter ([**?**]). Rather than construct the interface straightforwardly, we derive it indirectly from our original work by splitting the interface process into two parallel ones, which as well inspires us to obtain the complete set of partitioning rules from that in [**?**] without redefining them from scratch.

4.1. Protocol Converters

We define several protocol converters in this subsection, one of which will be adopted as the interface component between hardware and software within our partition framework.

First of all, we denote six sets that will be used frequently in following discussions:

sets of channel names: $B = \{cr_i, ca_i \mid i \in I\}$, and $B' = \{cr'_i, ca'_i \mid i \in I\}$, and

sets of signal variables: $E = \{er_i, ea_i \mid i \in I\} \cup \{ter\}$, and $E' = \{er'_i, ea'_i \mid i \in I\} \cup \{ter'\}$, and

sets of data variables: $V = \{x_i, y_i \mid i \in I\}$, and $V' = \{x'_i, y'_i \mid i \in I\}$.

The protocol converters are defined as follows.

Definition 4.1 (Protocol Converters) *The process c–c(B, B') is a converter between two communicating protocols on B and B', respectively:*

$$c\text{–}c(B, B') =_{df}$$
$$\mu X \bullet \left(\begin{array}{l} \|_{i\in I}(cr_i\,?\,x_i; cr'_i\,!\,x_i; ca'_i\,?\,y_i; ca_i\,!\,y_i; X) \\ \|\; skip \end{array} \right)$$

Similarly, the process s–s(E, E') converts a signal-driven protocol concerned with signals on E into that with respect to signals on E':

$$s\text{–}s(E, E') =_{df}$$
$$\mu X \bullet \left(\begin{array}{l} \|_{i\in I}(@er_i; \rightarrow er'_i; @ea'_i; \rightarrow ea_i; X) \\ \|\; (@ter; \rightarrow ter') \end{array} \right)$$

The two protocol converters to appear convert between protocols on two different levels. The process c–s$_V(B, E)$ converts a communicating protocol on B into a signal-driven protocol on E, whereas the process s–c$_V(E, B)$ is the reverse. The data variables from V are used to hold the information delivered by communication events:

$$c\text{–}s_V(B, E) =_{df}$$
$$\mu X \bullet \left(\begin{array}{l} \|_{i\in I}(cr_i\,?\,x_i; \rightarrow er_i; @ea_i; ca_i\,!\,y_i; X) \\ \|\; (skip; \rightarrow ter) \end{array} \right)$$

$$s\text{–}c_V(E, B) =_{df}$$
$$\mu X \bullet \left(\begin{array}{l} \|_{i\in I}(@er_i; cr_i\,!\,x_i; ca_i\,?\,y_i; \rightarrow ea_i; X) \\ \|\; (@ter; skip) \end{array} \right) \qquad \Box$$

Theorem 4.2 (Compositionality of Protocol Converters) *Each of the protocol converters mentioned above is a parallel composition of other two.*

(1). c–c(B, B') = c–s$_V(B, E)$ $\|$ s–c$_V(E, B')$, provided all variables in E and V are local ones only shared by c–s$_V(B, E)$ and s–c$_V(E, B')$;

(2). s–s(E, E') = s–c$_V(E, B)$ $\|$ c–s$_V(B, E')$, provided all channels in B are local ones only shared by s–c$_V(E, B)$ and c–s$_V(B, E')$;

(3). c–s$_V(B, E)$ = c–c(B, B') $\|$ c–s$_V(B', E)$, provided channels in B' are only used to perform communication between c–c(B, B') and c–s$_V(B', E)$;

(4). s–c$_V(E, B)$ = s–s(E, E') $\|$ s–c$_V(E', B)$, provided variables in E' are only shared by s–s(E, E') and s–c$_V(E', B)$. $\Box$

Theorem **??** explores the compositionality of protocol converters. As a by-product, it inspires us the way to derive the hybrid interface from the original one, and as well to

Figure 2. Protocol converter and interface derivation

compose the new hardware component under Verilog from that under $\mathcal{L}$.

Theorem 4.3 (Hardware Derivation) *The hardware component on different levels (defined in last two sections) can be derived from one another by protocol converters.*

(1). $D_V^B = c\text{–}s_V(B, E) \parallel D_V^E$, provided signal variables from E are local ones only shared by $c\text{–}s_V(B, E)$ and D_V^E;

(2). $D_{V'}^E = s\text{–}c_V(E, B') \parallel D_{V'}^{B'}$, provided channels in B' are only used by $s\text{–}c_V(E, B')$ and $D_{V'}^{B'}$ to arrange their interactions. $\square$

Theorem **??** shows that the new hardware component $D_{V'}^E$ in Verilog HDL can be constructed from the communicating one $D_{V'}^{B'}$ and a protocol converter. As well it reveals the way to generate the partitioning rules directly from the original ones. The interface and hardware derivation process is shown in Figure 2, where *c–s* denotes the new interface component, the dotted-box represents the new hardware component which hides those channels from B'.

4.2. Algebraic Proofs

We present the proofs for theorem **??** and **??** using program algebra in this subsection.

In addition to the algebraic laws mentioned in [**?**, **?**, **?**], we investigate the following laws, which will be used in the proofs later on.

The first law is an expansion law for the hybrid programming language([**?**]).

HL1 (*hgc-expan*, hybrid guarded choice expansion)
Let $G_1 = [\!]_{i\in I}(g_i; P_i)$, and $G_2 = [\!]_{j\in J}(h_j; Q_j)$, where g_i and h_j are of the form $ch\,!\,e$, or $ch\,?\,x$, or $skip$, or output event $\rightarrow ev$, or input event $@ev$, for all $i \in I$ and $j \in J$.

Let g and h denote the disjunction of all input event guards in G_1 and G_2, respectively. Then
$G_1 \parallel G_2 = [\!]_{r=1}^{n}(k_r; R_r)$, where the pairs $\langle k_r, R_r\rangle$ are precisely all possibilities from the following:

(1) $R_r = P_i \parallel G_2$, and
$k_r = g_i = skip$ or $k_r = g_i = ch\,!\,e$ or $k_r = g_i = ch\,?\,x$, where $ch \notin Chan(G_2)$, or
$k_r = g_i = \rightarrow ev$, or
$k_r = g_i \wedge \neg h$, and $g_i = @ev$

(2) $R_r = G_1 \parallel Q_j$, and
$k_r = h_j = skip$ or $k_r = h_j = ch\,!\,e$ or $k_r = h_j = ch\,?\,x$, where $ch \notin Chan(G_1)$, or
$k_r = h_j = \rightarrow ev$, or
$k_r = h_j \wedge \neg g$, and $h_j = @ev$

(3) $R_r = P_i \parallel Q_j$, and
$k_r = g_i \wedge h_j$, and $g_i = @ev$ and $h_j = @eu$

(4) $R_r = x := e; (P_i \parallel Q_j)$, and $k_r = skip$, and
$g_i = ch\,!\,e$, and $h_j = ch\,?\,x$, or
$g_i = ch\,?\,x$, and $h_j = ch\,!\,e$ $\square$

The second and third laws deal with two special cases with regard to signals .

HL2 (*eg-elim*, event guards elimination)
Let $P = G\,[\!]\,(@ev; Q)$, where G is a guarded choice, and ev is not a shared variable between P and its environment, then $P = G$. $\square$

HL3 (*self-trig*)
Let $P = \rightarrow ev; ((@ev; Q)\,[\!]\,G)$, where ev is not shared by the environment of P, and none of guards in G mentions ev, then $P = skip; Q$. $\square$

The following lemma is needed later in the proof of theorem **??**.

Lemma 4.4 *Let $P = M; P_1$, where $M \in MP(V)$, and G be a guarded choice construct not containing output guards of the kind $\rightarrow e$, where all guards in G are only shared by P and G, then we have*

$$P \parallel G = M; (P_1 \parallel G). \quad \square$$

It can be proved by a simple structural induction on M.

Now we can prove the theorems mentioned in last subsection.

The proof of theorem ??

(1). For brevity, we denote $c\text{–}s_V(B, E)$ as *c–s* and $s\text{–}c_V(E, B')$ as *s–c*.

RHS

$\{(hgc\text{-}expan),\ (eg\text{-}elim)\}$

$= [\!]_{i\in I}(cr_i\,?\,x_i; ((\rightarrow er_i; @ea_i; ca_i\,!\,y_i;\ c\text{–}s \parallel s\text{–}c))\,[\!]\,skip$

$\{(hgc\text{-}expan),\ (eg\text{-}elim),\ (self\text{-}trig)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,skip;\quad ((@ea_i;\,ca_i\,!\,y_i;\,c\text{–}s) \quad \| \quad (cr'_i\,!\,x_i;\,ca'_i\,?\,y_i;\,\rightarrow ea_i;\,s\text{–}c)))\ [\!]\ skip$

$\{(hgc\text{-}expan),\ (eg\text{-}elim)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,skip;\,cr'_i\,!\,x_i;\quad ((@ea_i;\,ca_i\,!\,y_i;\,c\text{–}s) \quad \| \quad (ca'_i\,?\,y_i;\,\rightarrow ea_i;\,s\text{–}c)))\ [\!]\ skip$

$\{(hgc\text{-}expan),\ (eg\text{-}elim)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,cr'_i\,!\,x_i;\,ca'_i\,?\,y_i;\quad ((@ea_i;\,ca_i\,!\,y_i;\,c\text{–}s) \quad \| \quad (\rightarrow ea_i;\,s\text{–}c)))\ [\!]\ skip$

$\{(hgc\text{-}expan),\ (eg\text{-}elim),\ (self\text{-}trig)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,cr'_i\,!\,x_i;\,ca'_i\,?\,y_i;\,skip;\quad ((ca_i\,!\,y_i;\,c\text{–}s) \quad \| \quad s\text{–}c))\ [\!]\ skip$

$\{(hgc\text{-}expan),\ (eg\text{-}elim)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,cr'_i\,!\,x_i;\,ca'_i\,?\,y_i;\,skip;\,ca_i\,!\,y_i;\quad (c\text{–}s \quad \| \quad s\text{–}c))\ [\!]\ skip$

$\{unique\ fixed\ point\ for\ guarded\ recursion\}$

$=\ LHS$

(2)-(4). Similar to (1). $\square$

The proof of theorem ??

(1). For brevity, we denote $c\text{–}s_V(B,E)$ as $c\text{–}s$.

RHS

$\{(hgc\text{-}expan),\ (eg\text{-}elim)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\quad ((\rightarrow \quad er_i;\,@ea_i;\,ca_i!y_i;\,c\text{–}s) \quad \| \quad D^E_V))\ [\!]\ (skip;\,(\rightarrow ter\ \|\ D^E_V)))$

$\{(hgc\text{-}expan),\ (eg\text{-}elim),\ (self\text{-}trig)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,skip;\quad ((@ea_i;\,ca_i!y_i;\,c\text{–}s) \quad \| \quad (M_i(x_i,y_i);\,\rightarrow ea_i;\,D^E_V)))\ [\!]\ (skip;\,skip)$

$\{Lemma\ \textbf{??}\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,skip;\,M_i(x_i,y_i);\quad ((@ea_i;\,ca_i!y_i;\,c\text{–}s) \quad \| \quad (\rightarrow ea_i;\,D^E_V)))\ [\!]\ skip$

$\{(hgc\text{-}expan),\ (eg\text{-}elim),\ (self\text{-}trig)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,M_i(x_i,y_i);\,skip;\quad ((ca_i!y_i;\,c\text{–}s) \quad \| \quad D^E_V))\ [\!]\ skip$

$\{(hgc\text{-}expan),\ (eg\text{-}elim)\}$

$= \quad [\!]_{i\in I}\,(cr_i\,?\,x_i;\,M_i(x_i,y_i);\,skip;\,ca_i!y_i;\quad (c\text{–}s \quad \| \quad D^E_V))\ [\!]\ skip$

$\{unique\ fixed\ point\ for\ guarded\ recursion\}$

$=\ LHS$

(2). Similar to (1). $\square$

5. Partitioning Rules in the Hybrid Framework

This section aims to generate hardware/software partitioning rules within the hybrid framework from those ones in [**?**] by using protocol converters.

To specify the partitioning rules, we introduce a new predicate $Split^E_B(S, C, c\text{–}s_V, D^E_V)$ which is defined by:

$$Split^E_B(S, C, c\text{–}s_V, D^E_V)\ =_{df}\ Split_B(S, C, (c\text{–}s_V(B,E)\ \|\ D^E_V))$$

where the predicate $Split_B$ is defined in [**?**] and is explained in section 2.

The following lemma is directly from theorem **??**.

Lemma 5.1 *We have*

$Split_B(S, C, D^B_V)$ **implies** $Split^E_B(S, C, c\text{–}s_V, D^E_V)$. $\square$

Based on this lemma, we generate all *bottom-up* rules straightforwardly as follows.

Bottom-up Rule for Sequential Composition

$$\frac{\begin{array}{c} Split^E_B(S_i,\ C_i,\ c\text{–}s_V,\ D^E_V),\ i=1,2 \\ Var(S_1)=Var(S_2),\ Chan(C_1)=Chan(C_2) \end{array}}{Split^E_B(S_1;S_2,\ C_1;C_2,\ c\text{–}s_V,\ D^E_V)}$$

Bottom-up Rule for Conditional

$$\frac{\begin{array}{c} Split^E_B(S_i,\ C_i,\ c\text{–}s_V,\ D^E_V),\ i=1,2 \\ Var(S_1)=Var(S_2),\ Chan(C_1)=Chan(C_2) \\ Var(b)\subseteq Var(C_1) \end{array}}{Split^E_B(if\ b\,S_1\ else\,S_2,\ if\ b\,C_1\ else\,C_2,\ c\text{–}s_V,\ D^E_V)}$$

Bottom-up Rule for Iteration

$$\frac{Split^E_B(S,\ C,\ c\text{–}s_V,\ D^E_V),\ Var(b)\subseteq Var(C)}{Split^E_B(while\,b\,S,\ while\,b\,C,\ c\text{–}s_V,\ D^E_V)}$$

When $Var(b)\cap Var(D)\neq\emptyset$, we will introduce a local variable lb, and rewrite the conditional and iteration into the forms

var $lb\bullet(lb:=b;\ if\ lb\,S_1\ else\,S_2)$, and
var $lb\bullet(lb:=b;\ while\,lb\,(S;\ lb:=b))$
respectively by the laws in [**?**].

Bottom-up Rule for Guarded Choice

$$\frac{\begin{array}{c} Split^E_B(S_i,\ C_i,\ c\text{–}s_V,\ D^E_V),\ i=1,2 \\ Var(S_1)=Var(S_2),\ Chan(C_1)=Chan(C_2) \\ Var(g_i)\subseteq Var(C_1),\ Chan(g_i)\subseteq Chan(C_1),\ i=1,2 \end{array}}{Split^E_B((g_1\,S_1)[\!](g_2\,S_2),\ (g_1\,C_1)[\!](g_2\,C_2),\ c\text{–}s_V,\ D^E_V)}$$

Similarly, we can obtain the *top-down* partitioning rules directly from those in [**?**] and the rules for primitive commands.

6 Conclusion

Hardware/software partition is a critical phase in hardware/software co-design. This paper proposes a hybrid partitioning framework, in which we design a set of protocol converters to construct the interface component between the hardware and software components and reuse those well-built partitioning rules in [**?**] by introducing protocol converters and rewriting them in the new framework. The hardware components our partitioning process generates are coded directly in Verilog HDL, which will greatly facilitate the further compilation from it down to netlists.

As further work, we shall reconfigure and reorganize the hardware component so as to optimize it and reuse its sub-components. Certain algebraic transformations shall be conducted to link our hardware component with the hardware specification to be compiled in [**?**] using Verilog algebra.

References

[1] A. Balboni *et al*, "Partitioning and Exploration Strategies in the TOSCA Design Flow", In *Proceedings of Fourth International Workshop on Hardware/Software Codesign*, 62–69, IEEE Computer Society Press, (1996).

[2] Geoffrey Brown, Wayne Luk and John O'Leary, "Retargeting a Hardware Compiler using Protocol Converters", in *Formal Aspects of Computing*, 8: 209-237, 1996.

[3] T. Cheung, "A Multi-level Transformation Approach to Hardware/Software Co-design", In *Proceedings of Fourth International Workshop on Hardware/Software Codesign*, 10–17, (1996).

[4] He Jifeng, "Hybrid Parallel Programming and Implementation of Synchronised Communication", In *18th International Symposium on Mathematical Foundations of Computer Science*, *Lecture Notes in Computer Science 711*, 537–546, (1993).

[5] He Jifeng, I. Page and J. Bowen, "A Provable Hardware Implementation of Occam", *Lecture Notes in Computer Science* 711, 693–703, (1993).

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

[7] He Jifeng and Zhu Huibiao, "Formalising Verilog", in the proceedings of the IEEE conference ICECS'2K, Lebanon, December 2000.

[8] He Jifeng *et al*, "Provably Correct Systems", *Lecture Notes in Computer Science* 863, 288–335, (1994).

[9] He Jifeng, "Converting Programs into Delay Insensitive Circuits", Technical Report, Programming Research Group, Oxford University Computing Laboratory, 1994.

[10] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[11] Juliano Iyoda and He Jifeng, "Towards an Algebraic Synthesis of Verilog", UNU/IIST Research Report 218, Macao SAR, China, April, 2001.

[12] Ian Page and Wayne Luk, "Compiling Occam into FPGAs", in *FPGAs*, eds., Will Moore and Wayne Luk, 271-283, Abingdon EE&CS books, (1991).

[13] Qin Shengchao and He Jifeng, "An Algebraic Approach to Hardware/software Partitioning", in the proceedings of the IEEE conference ICECS'2K, the IEEE press, Lebanon, December 2000.

[14] Augusto Sampaio, "An Algebraic Approach to Compiler Design", *World Scientific*, (1997).

[15] L. Silva, A. Sampaio and E. Barros, "A Normal Form Reduction Strategy for Hardware/software Partitioning", *Formal Methods Europe (FME) 97, Lecture Notes in Computer Science, 1313*, (1997) 624-643.