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
UNU/IIST, P.O.Box 3058, Macau, China
jifeng@iist.unu.edu

Abstract

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 the formerly well-built partitioning rules by introducing protocol converters and rewriting them for this hybrid framework. The hardware components generated by our partitioning process are coded directly in Verilog HDL, which will greatly facilitate the further compilation from it down to netlists.

Keywords: Hardware/software partition, protocol converter, program algebra

1. Introduction

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 the specification into hardware and software parts. Meanwhile, an interface component should be adopted to deal with the interactions between these parts. Recently, some works have suggested the use of formal methods for the partitioning process [1, 3, 15]. Balboni et al. ([11]) adopt Occam as an internal model for the system exploration and partitioning strategy. Cheung ([3]) 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 [15], Silva et al. provide a formal strategy for carrying out the splitting phase automatically, and presents an correctness proof. However, their splitting phase delivers many simple processes, and it's rather difficult to cluster and join these small processes into the target hardware and software components. Furthermore, additional channels and local variables introduced in the splitting to accommodate huge number of parallel processes actually increase the data flow between the hardware and software components.

Our former work ([13]) proposes a formal partitioning approach to hardware/software partition within a high-level communicating language, where both the hardware and software components are coded in the language. Due to this fact, an obvious gap exists in compiling the hardware specification down to netlists. In this paper, we'll bridge this gap by partitioning the system's specification into a software component coded in a high-level communicating language, a hardware component coded in a hardware description language (such as, Verilog), and as well an interface component to manage the interactions between hardware and software. A hardware component of this form will facilitate hardware compilation. Rather than construct all these components and the partitioning rules from scratch and then verify them in program algebra, we define a set of protocol converters (inspired by [2]), from which we obtain the interface component. Moreover, we derive new partitioning rules directly from those ones in [13] by introducing protocol converters and rewriting them. We prove the correctness of the protocol converters by the algebra for hybrid programming languages ([4]), which ensures the correctness of all the new partitioning rules.

*Partially supported by NNSFC No. 69873003
†on leave from East China Normal University

0-7695-1287-9/01 $17.00 © 2001 IEEE
The algebraic approach advocated in this paper has been successfully employed in the ProCos project on "Provably Correct Systems" [8, 6]. Sampaio showed how to reduce the compiler design task to one of program transformation by program algebra [14]. 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 [12], and He Jifeng et al provided a formal verification of the hardware compilation scheme within the algebra of Occam programs [5].

The remainder of this paper is organized as follows. Section 2 retrospects the original hardware/software partitioning framework and points out those parts that can be reused in the work presented here. Section 3 proposes the hybrid partitioning framework and poses the underlying architecture. In section 4, we define the protocol converters and provide the theoretical foundation for them. Section 5 shows how to gain the new partitioning rules from the original ones. A simple conclusion is presented in section 6.

2. Original Framework for Hardware/software Partitioning

This section is a brief introduction to our approach on hardware/software partitioning proposed in [13], 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:
   \[ S ::= PC \text{ (primitive command)} \]
   \[ | S, S \text{ (sequential composition)} \]
   \[ | \text{if } b \text{ then } S \text{ else } S \text{ (conditional)} \]
   \[ | S \cap S \text{ (non-deterministic choice)} \]
   \[ | \text{while } b \text{ do } S \text{ (iteration)} \]
   \[ | (g S) | (g S) \text{ (guarded choice)} \]

   where $PC ::= (x := e) | \text{skip} | \text{chaos} | e!e | e?x$ and $g$ is skip or a communication event $e!e$ or $d?x$.

2. Parallel Program:
   \[ P ::= S | P \parallel P | \text{var } x \bullet P \]

   In later discussions, we adopt $\text{Var}(P)$ and $\text{Chan}(P)$ 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 [9, 10] 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 $\text{CP}(B)$, for a given set of channels $B = \{cr_j, ca_j | j \in I\}$, where the set $\text{CP}(B)$ is the minimal set generated by the following rules:

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

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

   (3). $C_1; C_2$, or $C_1 \cap C_2$, or if $b \text{ then } C_1 \text{ else } C_2$, or $(g_1; C_1)(g_2; C_2)$, where both $g_1$ and $C_i$ lie in $\text{CP}(B)$, for $i = 1, 2$.

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

   The hardware component is chosen from the set of processes $\text{Hv}(B')$, for a given set of channels $B' = \{cr_j', ca_j' | j \in I\}$, and the set of data variables $V = \{x_j, y_j | j \in I\}$. The set $\text{Hv}(B')$ is composed by the processes of the form:

   \[ D_{B'}^H = \mu X \bullet \sum_{j \in I} (cr_j'^!x_j; M_j; ca_j'^!y_j; X) \text{skip} \]

   where none of $M_j$ mentions channels in $B'$.

   The communicating process $D_{B'}^H$ represents a digital device which offers a set of services, each of which responds to a request.
from its environment on an input channel \( cr'_i \) by running the corresponding program \( M_j \) and delivering the result to the output channel \( ca'_i \) afterwards.

To simplify the interface design, in [13], 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 \( H_Y(B') \), 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

\[
\mu X \cdot (\bigwedge_i \{ cr_i \; ? \; x_i; cr'_i \; ! \; x'_i; ca_i \; ? \; y_i; ca'_i \; ! \; y'_i; X \} \downarrow \text{skip})
\]

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 some as follows.

**Bottom-up Rule for Sequential Composition**

\[
\begin{align*}
\text{Split}_B(S_1, C_1, D_i, i = 1, 2) \\
\text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{Split}_S(S_1; S_2, C_1, C_2, D_i)
\end{align*}
\]

**Bottom-up Rule for Conditional**

\[
\begin{align*}
\text{Split}_B(S_1, C_1, D_i, i = 1, 2) \\
\text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{Var}(b) \subseteq \text{Var}(C_1) \\
\text{Split}_S(b \; S_1 \; \text{else} \; S_2, b \; C_1 \; \text{else} \; C_2, D_i)
\end{align*}
\]

**Bottom-up Rule for Iteration**

\[
\begin{align*}
\text{Split}_B(S, C, D) \\
\text{Var}(b) \subseteq \text{Var}(C)
\end{align*}
\]

**Bottom-up Rule for Non-deterministic Choice**

\[
\begin{align*}
\text{Split}_B(S_1, C_1, D_i, i = 1, 2) \\
\text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{Split}_S(S_1 \; \cap \; S_2, C_1 \; \cap \; C_2, D_i)
\end{align*}
\]

**Top-down Rule for Sequential Composition**

\[
\begin{align*}
\text{Split}_{B \cup B'}(S_1; S_2, C_1; C_2, \text{union}(D_1, D_2))
\end{align*}
\]

**Top-down Rule for Conditional**

\[
\begin{align*}
\text{Split}_{B \cup B'}(S_1, C_1, D_i, i = 1, 2) \\
\text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(S_1) = \text{Chan}(S_2) \\
\text{consist}(D_1, D_2) \\
\text{Var}(b) \subseteq \text{Var}(C_1) \\
\text{Split}_{B \cup B'}(b \; S_1 \; \text{else} \; S_2, b \; C_1 \; \text{else} \; C_2, \text{union}(D_1, D_2))
\end{align*}
\]

where the predicate \( \text{Split}_B(S, C, D) \) is defined in [13], which specifies that the process with \( C \) and \( D \) in parallel is a refinement of the specification \( S \), the predicate \( \text{consist}(D_1, D_2) \) 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. \( \text{union}(D_1, D_2) \) 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 [13]. 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 \( L \) 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([4]), 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 \( L \) defined in Section 2.

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 Section 2. It will be a specially chosen process from Verilog HDL, rather than a process in \( L \).

Given a set of signal variables \( E = \{ er_{i}, ea_{i} \mid i \in I \} \cup \{ \text{ter} \} \), and a set of data variables \( V = \{ x_{i}, y_{i} \mid i \in I \} \), we define a set of processes \( D(E, V) \), which is composed by processes of the following form:

\[
D_{E}^{E} =_{df} \mu X \bullet \left( \prod_{i \in I} ( \text{@er}_{i} \rightarrow \text{ea}_{i}; M_{i}(x_{i}, y_{i}); \rightarrow \text{ea}_{i}; X) \right) \left( \text{@ter}; \text{skip} \right)
\]

where \( \text{@er}_{i} \) and \( \rightarrow \text{ea}_{i} \) are respectively an input signal and an output signal in Verilog. The signal \( \text{ter} \) is used for synchronization 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 \( M \prod(V) \), which is constructed by the following rules:

1. \( v := e, \text{or skip, or chaos} \), where no signal variables from \( E \) occur in \( v \) or \( e \);
2. \( M; M' \), or \( \text{if b M else M'} \), or \( \text{while b M} \), where \( M, M' \in M \prod(V) \).

The signal-driven process \( D_{E}^{E} \) represents a digital device which offers a set of services to its environment, each of them responds to a request from its environment via an input signal \( er_{i} \) by running the corresponding program \( M_{i} \), then putting the result to variable \( y_{i} \) and delivering an acknowledgement through the output signal \( ea_{i} \) 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 ([22]). 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 [13] 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: \( B = \{ er_{i}, ea_{i} \mid i \in I \} \), and \( B' = \{ er_{i}', ea_{i}' \mid i \in I \} \);

sets of signal variables: \( E = \{ er_{i}, ea_{i} \mid i \in I \} \cup \{ \text{ter} \} \), and \( E' = \{ er_{i}', ea_{i}' \mid i \in I \} \cup \{ \text{ter}' \} \);

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:

\[
c-c(B, B') =_{df} \mu X \bullet \left( \prod_{i \in I} ( \text{@er}_{i}; \rightarrow \text{er}_{i}; \text{ea}_{i}; \rightarrow \text{ea}_{i}; X) \right) \left( \text{@ter}; \text{skip} \right)
\]

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

\[
s-s(E, E') =_{df} \mu X \bullet \left( \prod_{i \in I} ( \text{@er}_{i}; \rightarrow \text{er}_{i}; \text{ea}_{i}; \rightarrow \text{ea}_{i}; X) \right) \left( \text{@ter}; \text{skip} \right)
\]

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:

\[
c-s_{v}(B, E) =_{df} \mu X \bullet \left( \prod_{i \in I} ( \text{@er}_{i}; \rightarrow \text{er}_{i}; \text{ea}_{i}; \rightarrow \text{ea}_{i}; X) \right) \left( \text{@ter}; \text{skip} \right)
\]

\[
s-c_{v}(E, B) =_{df} \mu X \bullet \left( \prod_{i \in I} ( \text{@er}_{i}; \rightarrow \text{er}_{i}; \text{ea}_{i}; \rightarrow \text{ea}_{i}; X) \right) \left( \text{@ter}; \text{skip} \right)
\]

**Theorem 4.2 (Compositionality of Protocol Converters)**

Each of the protocol converters mentioned above is a parallel composition of other two.

1. \( c-c(B, B') = c-s_{v}(B, E) \parallel s-c_{v}(E, B') \), 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. \( s-s(E, E') = s-c_{v}(E, B) \parallel c-s_{v}(B, E') \), provided all channels in \( B \) are local ones only shared by \( s-c_{v}(E, B) \) and \( c-s_{v}(B, E') \);
3. \( c-s_{v}(B, E) = c-c(B, B') \parallel c-s_{v}(B', E) \), provided channels in \( B' \) are only used to perform communication between \( c-c(B, B') \) and \( c-s_{v}(B', E) \);
Figure 2. Protocol converter and interface derivation

(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 \( s^{-c}_V(E', B) \).

Theorem 4.2 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 compose the new hardware component under Verilog from that under \( E \).

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) \( D^B = c^{-s}_V(B, E) \ || \ D^E \), provided signal variables from \( E \) are local ones only shared by \( c^{-s}_V(B, E) \) and \( D^E \);

(2) \( D^B = s^{-c}_V(E, B) \ || \ D^E \), provided channels in \( B' \) are only shared by \( s^{-c}_V(E, B') \) and \( D^E \), to arrange their interactions.

Theorem 4.3 shows that the new hardware component \( D^B \) in Verilog HDL can be constructed from the communicating one \( D^E \) 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 \( c \)- notation 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 4.2 and 4.3 using program algebra in this subsection.

In addition to the algebraic laws mentioned in [13, 7, 4], 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([4]).

**HL1 (hgc-expansion, hybrid guarded choice expansion)**

Let \( G_1 = \bigcup_{i \in I} (g_i; P_i) \), and \( G_2 = \bigcup_{j \in J} (h_j; Q_j) \), where \( g_i \) and \( h_j \) are of the form \( \text{ch}! \, x \), or \( \text{ch}? \, x \), or \( \text{skip} \), 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 \( G_1 | G_2 = \bigcup_{i=1}^n (k_r; R_r) \), where the pairs \( (k_r, R_r) \) are precisely all possibilities from the following:

1. \( R_r = P_r \ || \ G_2, \) and \( k_r = g_i \ = \text{skip} \) or \( k_r = g_i = \text{ch}! \, x \), where \( ch \notin \text{Chan}(G_2) \), or \( k_r = g_i = @ev \), or \( k_r = g_i \wedge \neg h, \) and \( g_i = @ev \)

2. \( R_r = G_1 \ || \ Q_j, \) and \( k_r = h_j \ = \text{skip} \) or \( k_r = h_j = \text{ch}! \, x \), where \( ch \notin \text{Chan}(G_1) \), or \( k_r = h_j = @ev \), or \( k_r = h_j \wedge \neg g, \) and \( h_j = g_i = @ev \)

3. \( R_r = P_r \ || \ Q_j, \) and \( k_r = g_i \wedge h_j, \) and \( g_i = @ev \) and \( h_j = @ev \)

4. \( R_r = x := ev; (P_r \ || \ Q_j) \), and \( k_r = \text{skip}, \) and \( g_i = \text{ch}! \, x, \) and \( h_j = \text{ch}? \, x, \) or \( g_i = \text{ch} ? \, x, \) and \( h_j = \text{ch}! \, x \)

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

**HL2 (eg-elim, event guards elimination)**

Let \( P = G \backslash (e; ev; Q) \), where \( G \) is a guarded choice, and \( ev \) is not a shared variable between \( P \) and its environment, then \( P = G \).

**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 \( P = \text{skip}; Q \).

The following lemma is needed later in the proof of theorem 4.3.

**Lemma 4.4** Let \( P = M; P_0 \), where \( M \in MP(V) \), and \( G \) be a guarded choice construct not containing output guards of the kind \( \rightarrow ev \), where all guards in \( G \) are only shared by
\(P \text{ and } G, \text{ then we have}\)
\[
P \parallel G = M; (P_1 \parallel G).
\]
\[\square\]

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 4.2

(1). For brevity, we denote \(c-sv (B, E)\) as \(c-s\) and \(s-cv (E, B')\) as \(s-c\).

\[
\text{RHS}
\]

\[
\begin{align*}
\{ \text{hgc-expan}, & \text{ (eg-elim)} \} \\
\equiv & \int_{e \in E} (cr_i ? x_i; ((\rightarrow cr_i; @ea_i; ca_i ! y_i); c-s \parallel s-c)) \parallel \text{skip} \\
\{ \text{hgc-expan}, & \text{ (eg-elim), (self-trig)} \} \\
= & \int_{e \in E} (cr_i ? x_i; \text{skip}; ((@ea_i; ca_i ! y_i); c-s) \parallel (cr_i ! x_i; ca_i ! y_i \rightarrow ea_i; s-c)) \parallel \text{skip} \\
\{ \text{hgc-expan}, & \text{ (eg-elim)} \} \\
= & \int_{e \in E} (cr_i ? x_i; \text{skip}; cr_i ! x_i; ((@ea_i; ca_i ! y_i); c-s) \parallel (ca_i ? y_i \rightarrow ea_i; s-c)) \parallel \text{skip} \\
\{ \text{hgc-expan}, & \text{ (eg-elim), (self-trig)} \} \\
= & \int_{e \in E} (cr_i ? x_i; cr_i ! x_i; ca_i ! y_i; ((@ea_i; ca_i ! y_i); c-s) \parallel \rightarrow ea_i; s-c)) \parallel \text{skip} \\
\{ \text{hgc-expan}, & \text{ (eg-elim)} \} \\
= & \int_{e \in E} (cr_i ? x_i; cr_i ! x_i; ca_i ! y_i; \text{skip}; ((ca_i ! y_i; c-s) \parallel s-c)) \parallel \text{skip} \\
\{ \text{hgc-expan}, & \text{ (eg-elim), (self-trig)} \} \\
= & \int_{e \in E} (cr_i ? x_i; cr_i ! x_i; ca_i ! y_i; \text{skip}; ca_i ! y_i; (c-s \parallel s-c)) \parallel \text{skip} \\
\{ \text{unique fixed point for guarded recursion} \} \\
= & \text{LHS}
\end{align*}
\]

(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 [13] by using protocol converters.

To stipulate the partitioning rules, we introduce a new predicate \(\text{Split}_{\text{B}}(S, C, c-sv, D_{\text{B}}^F)\) which is defined by:
\[
\text{Split}_{\text{B}}(S, C, c-sv, D_{\text{B}}^F) = \text{if Split}_{\text{B}}(S, C, (c-sv (B, E) \parallel D_{\text{B}}^F))
\]
where the predicate \(\text{Split}_{\text{B}}\) is defined in [13] and is explained in section 2.

The following lemma is directly from theorem 4.3.

**Lemma 5.1** We have
\[
\text{Split}_{\text{B}}(S, C, D_{\text{B}}^F) \implies \text{Split}_{\text{B}}(S, C, c-sv, D_{\text{B}}^F).
\]

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

**Bottom-up Rule for Sequential Composition**
\[
\text{Split}_{\text{B}}^F(S_i, C_i, c-sv, D_{\text{B}}^F), \ i = 1, 2 \hspace{1cm} \text{Var}(S_1) = \text{Var}(S_2), \hspace{1cm} \text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{Split}_{\text{B}}^F(S_1 \parallel S_2, C_1, C_2, c-sv, D_{\text{B}}^F)
\]

**Bottom-up Rule for Conditional**
\[
\text{Split}_{\text{B}}^F(S_i, C_i, c-sv, D_{\text{B}}^F), \ i = 1, 2 \hspace{1cm} \text{Var}(S_1) = \text{Var}(S_2), \hspace{1cm} \text{Chan}(C_1) = \text{Chan}(C_2) \hspace{1cm} \text{Var}(b) \subseteq \text{Var}(C_1) \\
\text{Split}_{\text{B}}^F(\text{if } b S_1 \text{ else } S_2, C_1, C_2, c-sv, D_{\text{B}}^F)
\]

**Bottom-up Rule for Iteration**
\[
\text{Split}_{\text{B}}^F(S, C, c-sv, D_{\text{B}}^F), \text{Var}(b) \subseteq \text{Var}(C) \hspace{1cm} \text{Split}_{\text{B}}^F(\text{while } b \text{ S, while } b C, c-sv, D_{\text{B}}^F)
\]

146
When \( \text{Var}(b) \cap \text{Var}(D) \neq \emptyset \), we will introduce a local variable \( lb \), and rewrite the conditional and iteration into the forms

\[
\text{var } lb \bullet (lb := b; \text{ if } lb \in S_1 \text{ else } S_2), \text{ and } \text{var } lb \bullet (lb := b; \text{ while } (lb \in S_1) \text{ if } lb := b) \]

respectively by the laws in [13].

**Bottom-up Rule for Non-deterministic Choice**

\[
\text{Split}^E\{S_1, C_1, c\rightarrow v, D^E\}, i = 1, 2 \quad \text{Var}(S_1) = \text{Var}(S_2) \quad \text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{Split}^E\{S_1 \cap S_2, C_1 \cap C_2, c\rightarrow v, D^E\} 
\]

**Bottom-up Rule for Guarded Choice**

\[
\text{Split}^E\{S_1, C_1, c\rightarrow v, D^E\}, i = 1, 2 \quad \text{Var}(S_1) = \text{Var}(S_2) \quad \text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{Var}(g_i, y_j) \subseteq \text{Var}(C_1), \text{Chan}(g_i, y_j) \subseteq \text{Chan}(C_1), i = 1, 2 \\
\text{Split}^E\{[g_1, S_1], [g_2, S_2], (g_1, C_1)(g_2, C_2), c\rightarrow v, D^E\} 
\]

Before presenting the top-down partitioning rules, we introduce some notations and abbreviations which will be frequently used in the forthcoming rules.

Let

\[
E_i := \{e_{j, i, \in I_i} \cup \{\text{ter} \}, i = 1, 2 \\
B_i := \{e_{j, i, i} \cup \in I_i \}, i = 1, 2 \\
V_i := \{x_{j, i} \cup \in I_i \}, i = 1, 2
\]

For simplicity, from now on, we will always assume that

\[
E_1 \cap E_2 = \{\text{ter}\}, B_1 \cap B_2 = \emptyset, V_1 \cap V_2 = \emptyset.
\]

Given \( c\rightarrow v, B_i, E_i \), for \( i = 1, 2 \), we define

\[
c\rightarrow = \{c\rightarrow v, B_i \cup B_2, E_1 \cup E_2\}.
\]

Given

\[
D^E_{V_1} = \mu X \bullet \left( \biguplus_{i \in I_1} (\text{at} y, M_i(x_i, y_i) ; \rightarrow e_{a_i} ; X) \right),
\]

for \( i = 1, 2 \), we define

\[
\text{ExtVar}(D^E_{V_1}) := \{\text{Var}(D^E_{V_1}) \} \cup (E_i \cup V_i).
\]

\[
D = \text{merge}(D^E_{V_1}, D^E_{V_2}) := \mu X \bullet \left( \biguplus_{i \in I_1} \biguplus_{j \in I_2} (\text{at} y, M_i(x_i, y_i) ; \rightarrow e_{a_i} ; X) \right). \]

Now it’s time to present the top-down rules in our new partitioning framework.

**Top-down Rule for Sequential Composition**

\[
\text{Split}^E\{S_i, C_i, c\rightarrow v, D^E_{V_i}\}, i = 1, 2 \quad \text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{ExtVar}(D^E_{V_1}) = \text{ExtVar}(D^E_{V_2}) \\
\text{Split}^E\{S_1 \cup S_2, C_1 \cup C_2, c\rightarrow v, D^E\} 
\]

**Top-down Rule for Conditional**

\[
\text{Split}^E\{S_i, C_i, c\rightarrow v, D^E_{V_i}\}, i = 1, 2 \quad \text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{ExtVar}(D^E_{V_1}) = \text{ExtVar}(D^E_{V_2}) \\
\text{Var}(b) \subseteq \text{Var}(C_1) 
\]

**Top-down Rule for Non-deterministic Choice**

\[
\text{Split}^E\{S_i, C_i, c\rightarrow v, D^E_{V_i}\}, i = 1, 2 \quad \text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{ExtVar}(D^E_{V_1}) = \text{ExtVar}(D^E_{V_2}) \\
\text{Split}^E\{B_1 \cup B_2, (g_1, S_1) \cup (g_2, S_2), (g_1, C_1)(g_2, C_2), c\rightarrow v, D\} 
\]

**Top-down Rule for Guarded Choice**

\[
\text{Split}^E\{S_i, C_i, c\rightarrow v, D^E_{V_i}\}, i = 1, 2 \quad \text{Var}(S_1) = \text{Var}(S_2) \\
\text{Chan}(C_1) = \text{Chan}(C_2) \\
\text{ExtVar}(D^E_{V_1}) = \text{ExtVar}(D^E_{V_2}) \\
\text{Var}(g_i) \subseteq \text{Var}(C_1), \text{Chan}(g_i) \subseteq \text{Chan}(C_1), i = 1, 2 \\
\text{Split}^E\{B_1 \cup B_2, (g_1, S_1) \cup (g_2, S_2), (g_1, C_1)(g_2, C_2), c\rightarrow v, D\} 
\]

6 Conclusion

Hardware/software partitioning is a critical phase, and as well a challenging issue in hardware/software co-design. One possible way to cope with the specification of hardware, software and the interface between them separately, but this approach can hardly ensure that the composition of the three separate specifications satisfy the original requirement. In this sense, the co-specification of hardware, software and their interface has its superiority. In [13], we proposed a formal model for hardware/software partition within Occam algebra, in which a collection of provable partitioning rules have been explored. In that model, the interactions between hardware and software components appear very simple, due to the fact that they are both coded within the same high level communication language, which, on the other hand, reflects there exists a wide gap between our hardware component and the low level hardware description needed in later hardware synthesis phase.
In this paper, we refine the original partition approach and try to produce the hardware component coded directly in Verilog HDL. We adopt a hybrid partitioning framework, in which our partitioning process generates from the original system specification (a high level source program) three components: the software component in the high level communication language, the hardware component coded in Verilog HDL, and a hybrid interface component to cope with their interactions. Rather than construct partitioning rules in the new framework from scratch and then verify them, we introduce a set of protocol converters, from which we derive the new rules directly from the well-built ones in [13], and generate the new interface component together with the new hardware component.

Such an approach owns at least two merits: firstly, the original proved partitioning rules in our former model can be reused, and the proofs for the new rules become straightforward; secondly, the new hardware component is directly coded in Verilog HDL, which will facilitate the compilation from it down to the low level synthesizable description. However, the further synthesis of the interface component is still a challenging issue, which should be part of our future work. As other future 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 [11] using Verilog algebra.

References


