Protocol Compatibility and Converter Synthesis

Karin Avnit
kavnit@cse.unsw.edu.au
The university of NSW
Sydney, Australia

Vijay D'Silva
vdsilva@inf.ethz.ch
ETH
Zurich, Switzerland

Arcot Sowmya
sowmya@cse.unsw.edu.au
The university of NSW
Sydney, Australia

S. Ramesh
ramesh@cse.iitb.ac.in
Indian Institute of Technology
Bombay, India

Sri Parameswaran
sridevan@cse.unsw.edu.au
The university of NSW
Sydney, Australia

UNSW-CSE-TR-0625
Technical Report, December 2006

School of Computer Science and Engineering
The University of New South Wales
NSW 2052, Australia

Abstract

With increasing complexity of chip architecture and growing pressure for short time to market, hardware module reuse is common practice. However, in the absence of module interface standards, use of pre-designed modules in a "plug and play" fashion usually requires a mediator between mismatched interface protocols. Though several approaches to such mediation have been proposed, automation of protocol converter synthesis is yet to be realized. In this work we focus on the framework of state based protocol models. We present a formalism for modeling bus based communication protocols, the notion of compatibility and a protocol converter. Using this formalism, we provide algorithms for checking compatibility and demonstrate the process of automatic converter synthesis for commercial bus protocols.

1 Introduction

Aimed at accelerating the design phase and increasing system reliability, the use of pre-designed and pre-verified modules known as Intellectual Properties (IPs) for System-on-Chip (SoC) architecture is a natural choice. With this module reuse approach, a system can be built using modules that were developed separately and that will often endorse different interface protocols. For such modules to be able to communicate correctly, there is a need for unique glue logic (also referred to as transducer, converter, wrapper or bridge) to be introduced to mediate between them. A general bus-based SoC architecture is illustrated in Figure 1.

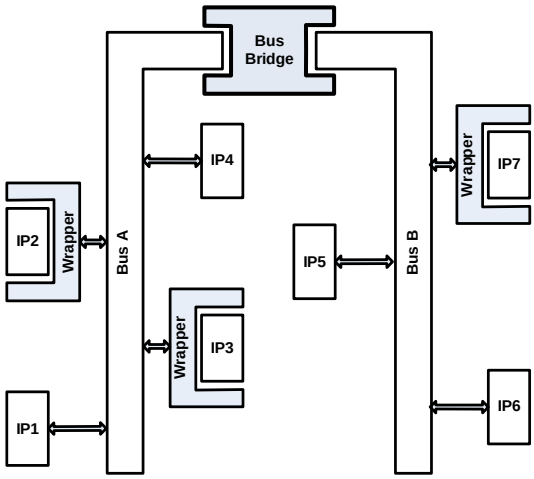

Figure 1: A typical SoC architecture

Much research has been dedicated to "the converter synthesis" problem of SoC communication, from attempts to standardize interface protocols [2,6,15,19], through methodologies for reusable design of IPs [17], research on the design of glue logic, some for specific protocols [5,8,9], some of a more general approach [10–13,16,20–23]. Attempts have been made to automate the process of generation of glue logic at different levels [1, 3, 4, 11, 12, 20–22].

Different approaches and models have been investigated for automation of converter synthesis: Timing diagrams [7], data queuing [13,23], message sequencing [22], FSM based protocol modeling [1, 3, 4, 11, 12, 21] and more.

In our work we focus on protocol compatibility and automatic synthesis of protocol converters in the framework of FSM based protocol modeling. We present a simple and powerful formal model for on-chip communication protocols that enables for the first time detailed modeling of complex commercial bus protocols and allows analysis of protocol compatibility and the automatic synthesis of a correct by construction converter at an abstraction level that is low enough to enable automatic translation to Hardware Description Language (HDL).

1.1 Related Work

The problem of automatically synthesizing a mediator for mismatched protocols has been addressed in the literature from different perspectives. We focus on work done

in the context of hardware design.

In early work [7] timing diagrams of the protocols were used as inputs and specification of a transducer was produced by constructing event graphs, requiring designers to provide information for correct merging of the graphs, with a requirement that data channels have the same name.

In [1] a protocol converter was constructed from a cross product of state machines that represent the protocols to be matched. The result was presented via a simple example. This approach was later extended in [3, 4, 12] and is the foundation of our work.

Later work [18] decomposes a sequential protocol into five basic operations and a protocol behavior is organized as ordered sets of guarded executions. An interface is then constructed by matching sets that transfer the same amount of data.

Mapping of any given protocol into a standard communication scheme is presented in [24], but the presented scheme requires that a protocol be either a sender or a receiver. The scheme can be applied in a multi-party communication environment but the solution is quite expensive, as there is a six-cycle latency between a data read and write and an internal arbiter is used that significantly increases the amount of logic in the system. This work was extended in [13, 23] by using protocol flow graph specifications to synthesize interfaces, which use queues and internal control logic to regulate buffering. The model requires the existence of specific channels and behaviors in the protocols to be matched.

All of the above mentioned work contributed to the evolution of solutions to the converter synthesis problem, but was preliminary, was not and probably could not be tested on realistic or commercial protocols.

An interface synthesis algorithm for mismatched synchronous protocols specified as regular expressions is provided in [21]. Their technique cannot be easily extended to different kinds of data and clock speed mismatches. In later work, Passerone et al. [20] stated that the above methods lack a mathematically sound formalization and attempted a game theoretic formalization. The synthesis procedure is defined as a winning strategy in a game held between a protocol and a converter and it is illustrated with an example that handles reordering of data. No algorithm is presented, so it is unclear how the proposed technique can be applied to any two arbitrary protocols.

A formalism for dealing with hardware interfaces as well as an algorithm for wrapper synthesis are proposed in [11,12]. The protocols in this work are represented as two FSMs and the outcome of the algorithm is a third FSM to be used as the wrapper for either of the protocols. Some methods for dealing with mismatched data types and clock periods are also presented though they are not integrated into the proposed algorithm.

Another approach presented in [22], focuses on a framework of message sequence charts. Its input is protocols represented as Message Sequence Charts, and the converter replies to messages from both protocols. The presented work handles only matched data types and it is not clear if it can handle complex protocols that have branching.

In [3,4] once again a product of an FSMs is used to construct a protocol converter, and in addition, the product is optimized for bandwidth.

The authors of [14] seem to have adopted the approach presented in [11,12] and in [14] they demonstrate its implementation over a great simplification of AMBA AHB and VCI BVCI.

Recent work [25] relies on [11,12] as well and shows how working in a higher level of abstraction reduces the size of models and therefore simplifies the generation of a converter at the cost of going further away from the desired hardware description converter.

In all previous work mentioned above, modeling of the protocols was done either in a high level of abstraction or with different levels of simplifications, and either way could not have result in complete automatic synthesis. The simple and powerful formalism that we propose in this work allows, for the first time, precise modeling at a low level of abstraction that can handle complete and complicated commercial protocols and can easily (and even automatically) be translated into HDL.

In vast majority of the discussed work, the definition of protocol compatibility is neglected or wrongly assumed to be trivial, and discussion of a protocol converter is introduced without a criteria as for when such a converter is needed. We propose for the first time a general and intuitive definition for protocol compatibility in the context of ensuring continuous data flow between two protocols. We then formalize the notion of compatibility and propose an algorithm for checking compatibility as well as a formal definition for the protocol converter synthesis problem.

1.2 Motivating Examples

Consider the following scenario: a slave designed to interface with an AMBA APB bus needs to be integrated to a system working with an AMBA ASB bus.

A timing diagram of a write operation of ASB is given in Figure 2 while a timing diagram of ABP is given in Figure 3 (both taken from [6]).

Other than the different Signal naming between the two protocols, ASB is a much more complicated protocol than ABP and requires a slaves that supports "slave response" which APB does not have. Clearly the two protocols are incompatible and a protocol converter needs to be introduced in order to have the module work correctly in the system.

We propose models for APB and ASB slave write operation interfaces as depicted in Figure 4, note that ASB is active on both edges of the clock while APB is active only on clock rise, which complicates the APB model, in the same way that different clock frequencies effect the model as suggested in [11]. The notation of the models is explained in details in section 2 but the incompatibility of the two models can be seen easily.

In ABP, a slave is idle as long as it is not selected. Once the $PSEL$ and $WRITE$ controls are high a write transfer of two clock cycles begins.

The ASB model represents the protocol that a slave needs to react to (the way a slave sees the behavior of the system). The system can stay idle until it chooses to initiate a write transfer by asserting the $DSEL$ and $BWRITE$ control, at the same time it puts an address on the bus as well as some other controls. Once the slave responses with a $DONE$ or $LAST$ the data to be written is put on the bus. (for more details on ASB we refer the reader to [6])

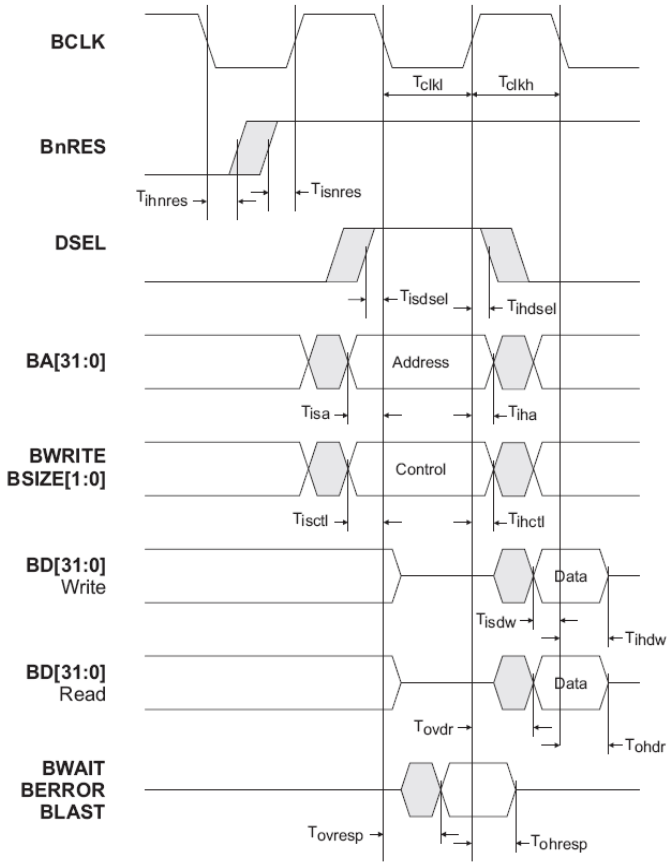

Figure 2: ASB slave timing diagram

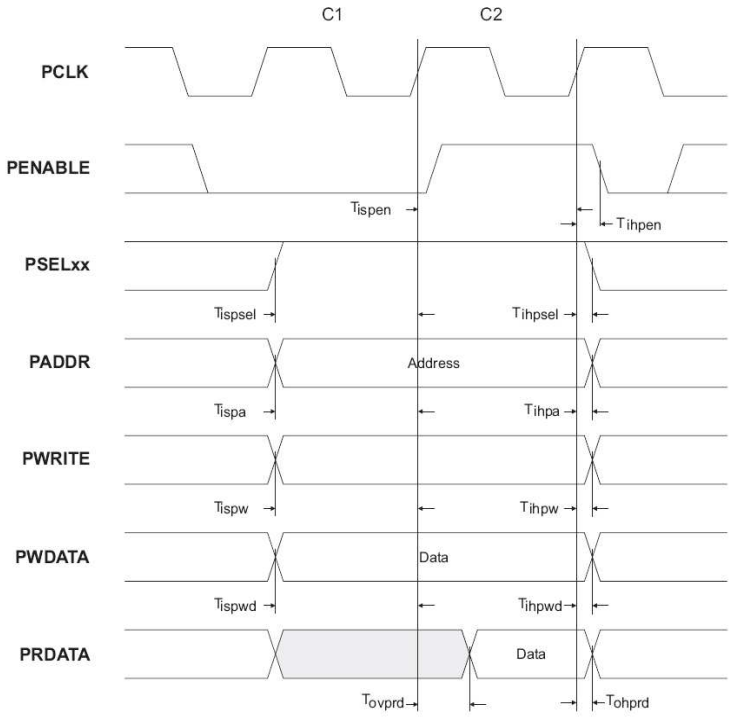

Figure 3: APB slave timing diagram

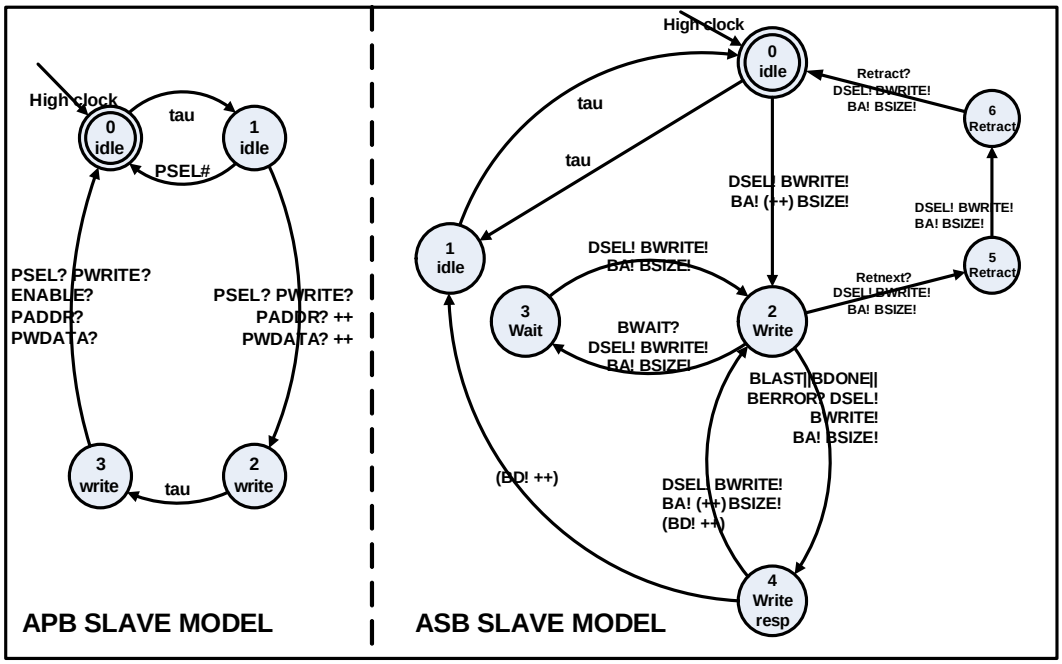

Figure 4: Protocol models

2 Formal Definitions

2.1 A Protocol

We model protocols as synchronous finite state machines with bounded counters that communicate using channels. Channels are of two types, control and data. A protocol can test for the existence or absence of a value on a control channel, denoted by $c?$ and $c\#$ respectively and write to a control channel, denoted by $c!$. A protocol can also read values from or write values to a data channel, denoted by $d?$ and $d!$ respectively. A *channel action* is a read, a write or a value test on a channel. Let A_Σ denote the set of possible actions on a set of channels Σ.

Bounded counters are used to monitor the number of data items written or read in a finite burst. A counter is associated with each data channel. The counter value is changed (that is, incremented or reset) when a new data is written to or read from a channel. Counters provide expressivity and brevity:

- Between two changes to the counter value, any read or write indicates data repetition. This feature could not previously be modeled.
- As many protocols support bursts of various lengths, explicitly representing them would result in a large FSM. Using bounded counters allows for smaller models.

Let K be a set of counters. The set of counter actions $A_K = \{reset(k), k{+}{+}, k = v | k \in K, v \in \mathbb{N}\}$ are a reset, increment or test for equality with some natural number. A protocol performs channel and counter actions.

Definition 1 (Protocol) *A protocol P is a finite state machine with bounded counters $(Q, C, D, K, \rightarrow, q_s, q_f)$, where Q is the set of states, $C = C^I \cup C^O$ is a set of input and output control channels, $D = D^I \cup D^O$ is a set of input and output data channels, $K = \{k_d | d \in D\}$ is a set of bounded internal counters with one for each data channel ($|K| = |D|$), q_s is the initial state and q_f is the final state. Let $A_P = A_C \cup A_D \cup A_K$ be the set of actions on the control channels (A_C), data channels (A_D) and counters (A_K) of P. The transition relation of the protocol is $\rightarrow \subseteq Q \times \mathcal{P}(A_P) \times Q$ (where $\mathcal{P}(A_P)$ is the powerset of A_P).*

2.2 Parallel Composition

We are interested in the behavior of protocols executing concurrently. This behavior is described by the parallel composition of the protocols. We define a binary predicate `may` to identify transitions that may occur together and use this predicate to define the parallel composition operator.

Definition 2 (`may`) *The predicate* $\texttt{may}(S_1, S_2)$ *is true for two actions S_1 and S_2 iff for every control channel $c \in C$:*

if $c? \in S_1$, then $c! \in S_2$, if $c\# \in S_1$, then $c! \notin S_2$,
if $c? \in S_2$, then $c! \in S_1$, if $c\# \in S_2$, then $c! \notin S_1$.

Definition 3 (Parallel Composition) *The parallel composition $P_1 \| P_2$ of two protocols $P_1 = (Q_1, C_1, D_1, K_1, \rightarrow_1, q1_s, q1_f)$ and $P_2 = (Q_2, C_2, D_2, K_2, \rightarrow_2, q2_s, q2_f)$ is a finite state machine with bounded counters $(Q_1 \times Q_2, (C_1 \cup C_2) \setminus (C_1 \cap C_2), (D_1 \cup D_2) \setminus (D_1 \cap D_2), K_1 \cup K_2, \rightarrow, (q1_s, q2_s), (q1_f, q2_f))$, where $(q1, q2) \xrightarrow{S} (q1', q2')$ is a transition of $P_1 \| P_2$ iff $q1 \xrightarrow{S_1}_1 q1'$ and $q2 \xrightarrow{S_2}_2 q2'$ are transitions of P_1 and P_2 respectively, such that $\texttt{may}(S_1, S_2)$ is true and S is the set of actions occurring in $S_1 \cup S_2$ excluding complementary actions.*

2.2.1 An example

Considering the two protocols P_1 and P_2 as presented in Figure 5, under a mapping of the channels $c1 \Leftrightarrow c2, a1 \Leftrightarrow a2, d1 \Leftrightarrow d2$, the parallel composition of the two protocols according to definition 3 is illustrated in Figure 6.

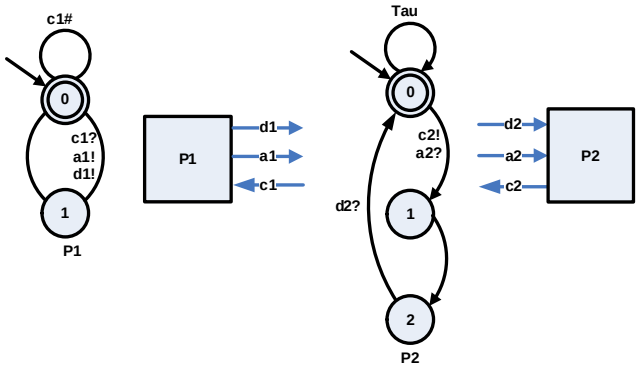

Figure 5: The protocols. c_i is a control channel, a_i and d_i are data channels

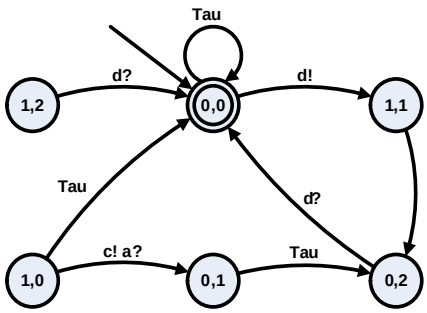

Figure 6: The parallel composition of the two protocols

2.3 Paths

A path in a protocol P is a sequence of states and transitions

$$\pi = q_0 \xrightarrow{S_1} q_1 \ldots \xrightarrow{S_k} q_k$$

such that for all $0 \leq i < k$, $q_i \xrightarrow{S_i} q_{i+1}$ is a transition in P.

- Let $Paths(P, q_j, q_k)$ denote the set of paths in P from state q_j to state q_k.
- Define $New(\pi, d) = \{i \in \mathbb{N} | 0 < i \leq k \text{ and } reset(k_d) \in S_i \text{ or } k_d{+}{+} \in S_i\}$ as the set of indices on a path at which new data is either written or read on channel d.
- Let $|\pi|$ denote the number of transitions in path π.
- For a path π in a finite state machine $P_1 \| P_2$, there exist paths π_1 and π_2 in P_1 and P_2 (implied by the definition of parallel composition).
- A *cycle* is a path such that $q_0 = q_k$.
- *equivalent cycles*: two cycles of length k, $\pi_1 = q1_0 \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_k} q1_0$ and $\pi_2 = q2_0 \xrightarrow{S2_1} q2_1 \ldots \xrightarrow{S2_k} q2_0$ are *equivalent cycles* iff for some integer ℓ, for $1 \leq i \leq k$: $q1_{i-1} \xrightarrow{S1_i} q1_i = q2_{(\ell+i-1) mod(k)} \overset{S2_{(\ell+i) mod(k)}}{\longrightarrow} q2_{(\ell+i) mod(k)}$.

2.3.1 Operations Over Paths

For two paths $\pi_1 = q1_0 \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_{k1}} q1_{k1}$ and $\pi_2 = q2_0 \xrightarrow{S2_1} q2_1 \ldots \xrightarrow{S2_{k2}} q2_{k2}$

- $\pi_1 \cap \pi_2$ is the set of states ($Q_{\pi_1 \cap \pi_2}$) and transitions ($\rightarrow_{\pi_1 \cap \pi_2}$) such that
  $q \in Q_{\pi_1 \cap \pi_2}$ iff $q \in \pi_1$ and $q \in \pi_2$, and
  $q_0 \xrightarrow{S} q_1 \in \rightarrow_{\pi_1 \cap \pi_2}$ iff $q_0 \xrightarrow{S} q_1 \in \pi_1$ and $q_0 \xrightarrow{S} q_1 \in \pi_2$.
- $\pi_1 + \pi_2$ is defined for the following cases:
  1. For π_1, π_2 such that $q1_{k1} = q2_0$
     $\pi_1 + \pi_2 = q1_0 \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_{k1}} q1_{k1} \xrightarrow{S2_1} q2_1 \ldots \xrightarrow{S2_{k2}} q2_{k2}$
     $(\pi_1 + \pi_2 \neq \pi_2 + \pi_1)$.
  2. For π_1, π_2 such that π_2 is a cycle and $\pi_1 \cap \pi_2 \neq \emptyset$, for a state $q_m = q1_i = q2_j \in \pi_1 \cap \pi_2$,
     $\pi_1 + \pi_2 = q1_0 \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_i} q_m \xrightarrow{S2_j} q2_{j+1} \ldots \xrightarrow{S2_{j-1}} q_m \xrightarrow{S1_{i+1}} q1_{i+1} \ldots \xrightarrow{S1_{k1}} q1_{k1}$
     (since $\pi_1 \cap \pi_2$ may have more than one state, $\pi_1 + \pi_2$ may result in more than one path)

  for all other cases $\pi_1 + \pi_2$ is not defined.
- $\alpha \times \pi$ (or $\alpha\pi$) is defined for cycles only and is the same as $\sum_{i=1}^{\alpha} \pi$.
- $\pi_1 \subseteq \pi_2$ is true iff all transitions of π_1 are in π_2 in the same order as they appear in π_1.

2.4 Constructing $Paths(P, q_s, q_f)$

For a general protocol P that has cycles, $Paths(P, q_s, q_f)$ is an infinite group, made of a finite set of direct (acyclic) paths and a set of cycles that can be added to the direct paths any number of times. The group can then be represented symbolically as a finite set of symbolic paths where for a path π_1 and a cycle π_2 the symbolic path $\pi_1 + \alpha\pi_2$ represents all paths made of π_1 and any number of repetitions of cycle π_2.

In order to extract the sets of direct paths and possible cycles, we use a representation similar to a computation tree, spanning every possible transition from the initial state (the root of the tree) such that every final state is a leaf of the tree and every state that already exists on its path from the root is a leaf. The tree is of finite depth ($\leq |Q|$)

In such a tree every final state leaf represents a direct path from initial to final state and every other leaf represents a cycle in the graph (it is possible that several leaves represent the same cycle in the graph). In a case where the initial and final states are not the same, an additional tree needs to be created for all paths beginning and ending in the final state and all of its paths should be added to the list of cycles.

2.4.1 An example

The computation tree of the ASB model from Figure 4 is presented in Figure 7

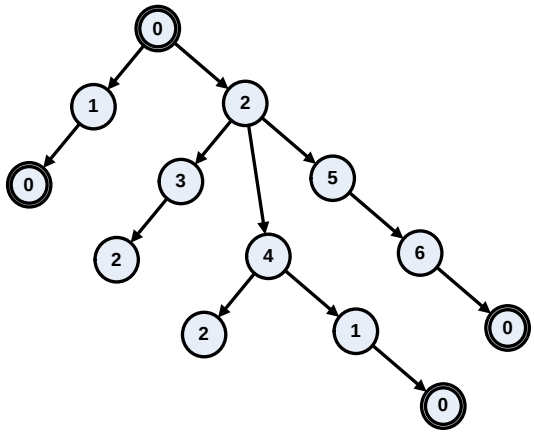

Figure 7: The computation tree for ASB

$$DirectPaths = \{ \quad \pi_1 = 0 \xrightarrow{Tau} 1 \xrightarrow{Tau} 0, \qquad \pi_2 = 0 \xrightarrow{S_1} 2 \xrightarrow{S_2} 4 \xrightarrow{S_3} 1 \xrightarrow{Tau} 0, \qquad \pi_3 = 0 \xrightarrow{S_1} 2 \xrightarrow{S_4} 5 \xrightarrow{S_5} 6 \xrightarrow{S_6} 0 \ \}$$

$$Cycles = \{ \quad \pi_4 = 2 \xrightarrow{S_7} 3 \xrightarrow{S_8} 2, \qquad \pi_5 = 2 \xrightarrow{S_2} 4 \xrightarrow{S_9} 2,$$

where:

S_1 = "DSEL! BWRITE! BA! (++) BSIZE!",

S_2 = "BLAST||BDONE||BERROR? SDEL! BWRITE! BA! BSIZE!"',

S_3 = "BD! (++)",

S_4 = "RETNEXT? DSEL! BWRITE! BA! BSIZE!",

S_5 = "DSEL! BWRITE! BA! BSIZE!",

S_6 = "RETRACT? DSEL! BWRITE! BA! BSIZE!",
S_7 = "BWAIT? DSEL! BWRITE! BA! BSIZE!",
S_8 = "DSEL! BWRITE! BA! BSIZE!".
S_9 = "DSEL! BWRITE! BA! (++) BSIZE! BD! (++)",

Once the two sets $DirectPaths$ and $Cycles$ are computed, the construction of $Paths(P, q_s, q_f)$ can be done as described in Algorithm 1:

Algorithm 1 $ComputePaths(P, q_s, q_f)$

```
Paths(P, q_s, q_f) = DirectPaths;
PendingStates = DirectPaths;
for all π ∈ PendingStates do
  i = 1;
  for all c in Cycles do
    if c ⊈ π and c ∩ π ≠ ∅ then
      add π + α_i c to Paths(P, q_s, q_f);
      add π + α_i c to PendingStates;
      i++ ;
    end if;
    remove π from PendingStates;
  end for;
end for;
```

And in our ASB example:

$$Paths(ASB, 0, 0) = \{\ \pi_1\ ,\ \pi_2\ ,\ \pi_3\ ,\ \pi_2 + \alpha_1\pi_4\ ,\ \pi_2 + \alpha_1\pi_5\ ,\ \pi_2 + \alpha_1\pi_4 + \alpha_2\pi_5\ ,\ \pi_3 + \alpha_1\pi_4\ ,\ \pi_3 + \alpha_1\pi_5\ ,\ \pi_3 + \alpha_1\pi_4 + \alpha_2\pi_5\ \}$$

$\alpha_i \in \mathbb{N}, > 0$

3 Compatibility

We focus on compatibility with respect to ensuring data flow between a pair of protocols. The parallel composition of two protocols describes the possible states they may be in when run concurrently. To ensure correct data flow between these protocols some constraints must be satisfied:

- Data is read by one protocol only when written by the other.
- A given data item is read exactly once.
- No undefined behaviors can be reached, and every transaction can terminate within finite time (equivalent to a finite number of transitions).

For notational simplicity, in the following definition, we assume that there is exactly one data channel d that is written to by P_1 and read from by P_2, as illustrated in Figure 8. We define compatibility in terms of constraints over paths in the parallel composition of two protocols.

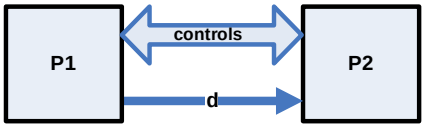

Figure 8: System structure

Definition 4 (Compatibility) *Two protocols P_1 and P_2 are compatible iff:*

1. *$Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f))$ is not empty.*
2. *For any path $\pi \in Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f))$ corresponding to the paths $\pi_1 = q1_s \xrightarrow{S_1^1} q1_1 \ldots \xrightarrow{S_k^1} q1_f$ in P_1 and $\pi_2 = q2_s \xrightarrow{S_1^2} q2_1 \ldots \xrightarrow{S_k^2} q2_f$ in P_2, the following hold:*
   (a) *if $d? \in S_i^2$ then $d! \in S_i^1$.*
   (b) *$|New(\pi_1, d)| = |New(\pi_2, d)|$.*
   (c) *Let $i_1 < i_2 \cdots < i_n$ be the sorted sequence of indices in $New(\pi_1, d)$ and $j_1 < j_2 \cdots < j_n$ be the sequence of sorted indices in $New(\pi_2, d)$. For indices ℓ it holds that $j_{\ell-1} < i_\ell \leq j_\ell < i_{\ell+1}$ where $1 \leq \ell \leq n$ and j_0 is defined as 0 and i_{n+1} is defined as $k+1$ (where k is the number of transitions in the path).*
3. *For any state $(q1, q2) \in P_1 \| P_2$ such that $Paths(P_1 \| P_2, (q1_s, q2_s), (q1, q2))$ is not empty, it holds that $Paths(P_1 \| P_2, (q1, q2), (q1_f, q2_f))$ is not empty.*

The first requirement for compatibility is that $Paths(P_1 \| P_2)$ is not an empty set. This ensures that the protocols can execute together from initial to final states. If the set of paths in $P_1 \| P_2$ is not empty, we require that every path from the initial to the final state should satisfy three conditions. (a) Only valid data is read. (b) The same number of distinct data items are written and read by the two protocols in any path to the final state. (c) The third condition ensures the first two constraints of correct data flow. A new data item is written only after the previous one has been read. The same data item is not treated as distinct data items if read multiple times. The last requirement is needed to guarantee the absence of undefined behaviors and the finite length of transactions - Every state that can be reached should have a path to the final state. In a general case where there is more than one data channel, condition 2 should hold for every channel independently.

3.1 Checking Compatibility

For two protocols P_1 and P_2 a compatibility check can be done as in Algorithm 2

In the example of $P1$ and $P2$ the parallel composition demonstrated in Figure 6 will fail the transition check, as the transition from state $(0, 2)$ to state $(0, 0)$ has a read operation that does not have a corresponding write operation clause 2a in the compatibility definition and line 3 in Algorithm 3, and therefore, the two protocols will be found to be incompatible.

Algorithm 2 $Compatible(P_1, P_2)$

Compute $P1 \| P2$; {include only reachable states and transitions}
if $CheckTransitions(P1 \| P2)$ == FALSE **then**
  return FALSE;
end if;
Compute $Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f))$;
if $Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f)) = \emptyset$ **then**
  return FALSE;
end if;
if (states of $P1 \| P2$) $\neq$ (states of $Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f))$) **then**
  return FALSE;
end if;
for all π (symbolic path) in $Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f))$ **do**
  if $CheckPath(\pi, P_1 \| P_2)$ == FALSE **then**
    return FALSE;
  end if;
end for;
return TRUE; {the protocols are compatible}

Algorithm 3 $CheckTransitions(P = (Q, C, D, K, \rightarrow, q_s, q_f))$

for all $t \in \rightarrow$ **do**
  for all $d \in D$ **do**
    if "d?" $\in t$ and "d!" $\notin t$ **then**
      return FALSE;
    end if;
  end for;
end for;
return TRUE; {all transitions have passed the check}

Algorithm 4 $CheckPath(\pi, P = (Q, C, D, K, \rightarrow, q_s, q_f))$

π is of the form $\pi_0 + \alpha_1\pi_1 + \ldots + \alpha_n\pi_n$
$CheckedPath = \pi_0 + \pi_1 + \ldots + \pi_n$; {no need to check for $\alpha > 1$}
$k = |CHeckedpath|$;
π_{P_1} is the projection of $CheckedPath$ on P_1
π_{P_2} is the projection of $CheckedPath$ on P_2
for all $d \in D$ **do**
  if d is input to P_2 **then**
    Compute $New(\pi_{P_1}, d) = \{i_1 < i_2 \cdots < i_{n1}\}$;
    Compute $New(\pi_{P_2}, d) = \{j_1 < j_2 \cdots < j_{n2}\}$;
  else
    {d is input to P_1}
    Compute $New(\pi_{P_1}, d) = \{j_1 < j_2 \cdots < j_{n1}\}$;
    Compute $New(\pi_{P_2}, d) = \{i_1 < i_2 \cdots < i_{n2}\}$;
  end if
  if $|New(\pi_{P_1}, d)| \neq |New(\pi_{P_2}, d)|$ **then**
    return FALSE;
  else
    $n = |New(\pi_{P_1}, d)|$;
    $j_0 = 0$;
    $i_{n+1} = k + 1$;
    for all ℓ such that $1 \leq \ell \leq n$ **do**
      something
      if $(j_{\ell-1} \geq i_\ell)$ or $(i_\ell > j_\ell)$ or $(j_\ell \geq i_{\ell+1})$ **then**
        return FALSE;
      end if
    end for;
  end if
end for;
return TRUE;

In the example of ASB and APB protocols, due to the severe incompatibility of the protocols, even with channel mapping of $PSEL \Leftrightarrow DSEL$, $PWRITE \Leftrightarrow BWRITE$, $PADDR \Leftrightarrow BA$, $PWDATA \Leftrightarrow BD$, the reachable component of the parallel composition is as illustrated in Figure 9. This would pass the transitions check but will fail the check that all reachable states can reach the final state, as in line 9 in Algorithm 2 (state $(2, 1)$ can be reached from the initial state but does not have a path to the the final state)

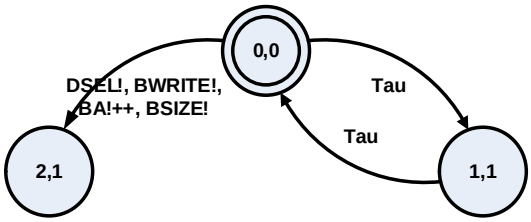

Figure 9: The parallel composition of ASB and APB

4 Converter Synthesis

If two protocols are incompatible, a converter has to be synthesized. In a similar manner to that of direct inter module communication, to ensure correct data flow between two protocols P_1, P_2 communicating through a converter some constraints must be satisfied:

- Data is read by one protocol (/the converter) only when written by the converter (/a protocol).
- A given data item is read exactly once.
- Every data item written by $P_1 (/P_2)$ to the converter will be written by the converter to P_2 $(/P_1)$.
- Only the protocols (P_1, P_2) can write new data items in the system. (i.e. every data item written by the converter was previously written by a protocol)
- No undefined behaviors can be reached, and every transaction can terminate within finite time (equivalent to a finite number of transitions).

For notational simplicity, in the following definitions, we assume that there is exactly one data channel d_1 that is written to by P_1 and read from by the converter and a corresponding channel d_2 that is written by the converter and read by P_2, as illustrated in Figure 10.

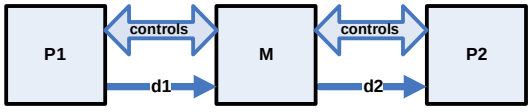

Figure 10: System structure

Definition 5 (Converter Synthesis Problem) *Given two incompatible protocols P_1 and P_2, synthesize a finite state machine with bounded counters M satisfying the following:*

1. *P_1 and M are compatible.*
2. *P_2 and M are compatible.*
3. *For any path $\pi_m \in Paths(M, qm_s, qm_f)$ the following hold:*
   (a) *$|New(\pi_m, d_1)| = |New(\pi_m, d_2)|$.*
   (b) *let $i_1 < i_2 \cdots < i_n$ be the sorted sequence of indices in $New(\pi_m, d1)$ and $j_1 < j_2 \cdots < j_n$ be the sequence of sorted indices in $New(\pi_m, d2)$. For indices ℓ it holds that $j_{\ell-1} < i_\ell \leq j_\ell < i_{\ell+1}$ where $1 \leq \ell \leq n$ and j_0 is defined as 0 and i_{n+1} is defined as $k+1$ (where k is the number of transitions in the path).*

The first two requirements guarantee that data items are read exactly once, only when written and no undefined behaviors can be reached (corresponding to the first, second and fifth constraints for correct data flow). The third requirement relates to a notion of fairness/robustness on behalf of the converter - guaranteeing that the converter passes all given data and does not make up data on its own (corresponding to the third and fourth constraints).
In a general case where there are more data channels, constraint 3 should hold for every pair of mapped channels independently.

4.1 Guidelines for the construction of a correct converter

Automatic construction of a protocol converter, given the models of the protocols P_1 and P_2 and data channels mapping information will include the following steps:

1. Construct a product finite state machine $M^\star = (Q_1 \times Q_2, (C_1 \cup C_2), (D_1 \cup D_2), K_1 \cup K_2, \rightarrow, (q1_s, q2_s), (q1_f, q2_f))$, where $(q1, q2) \xrightarrow{S} (q1', q2')$ is a transition of $P_1 \| P_2$ iff $q1 \xrightarrow{S_1}_1 q1'$ and $q2 \xrightarrow{S_2}_2 q2'$ are transitions of P_1 and P_2 respectively, channel directions are inverted and S is the set of actions complementary to all actions occurring in $S_1 \cup S_2$.

   $M^\star$ includes all possible pairs of transitions and therefore describes the most general behavior of the two protocols in parallel. Out of $M^\star$ we are interested only in states and transitions that can be reached in a run beginning at the initial state, and therefore construct $M = (Q_M, (C_1 \cup C_2), (D_1 \cup D_2), K_1 \cup K_2, \rightarrow, (q1_s, q2_s), (q1_f, q2_f))$, where state $(q_1, q_2) \in Q_M$ iff $Paths(M^\star, (q1_s, q2_s), (q_1, q_2))$ is not empty, and a transition $(q1, q2) \xrightarrow{S} (q1', q2')$ is a transition of M iff $(q1, q2) \in Q_M$.

2. Pruning of M while keeping M to be compatible with the protocols, to resolve any nondeterminism that is not derived from buffering aspects.

We propose the following converters to the examples given in this report: Figure 11 illustrate the converter for P_1 and P_2, while Figure 12 provides the model for AMBA ASB to AMBA APB slave protocol converter, for write operations.

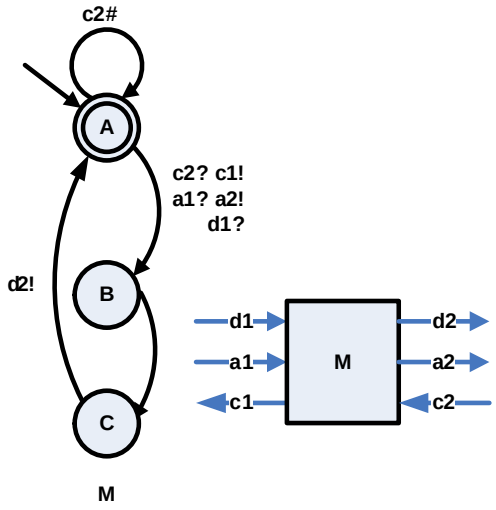

Figure 11: A suggested converter for protocols P1 and P2

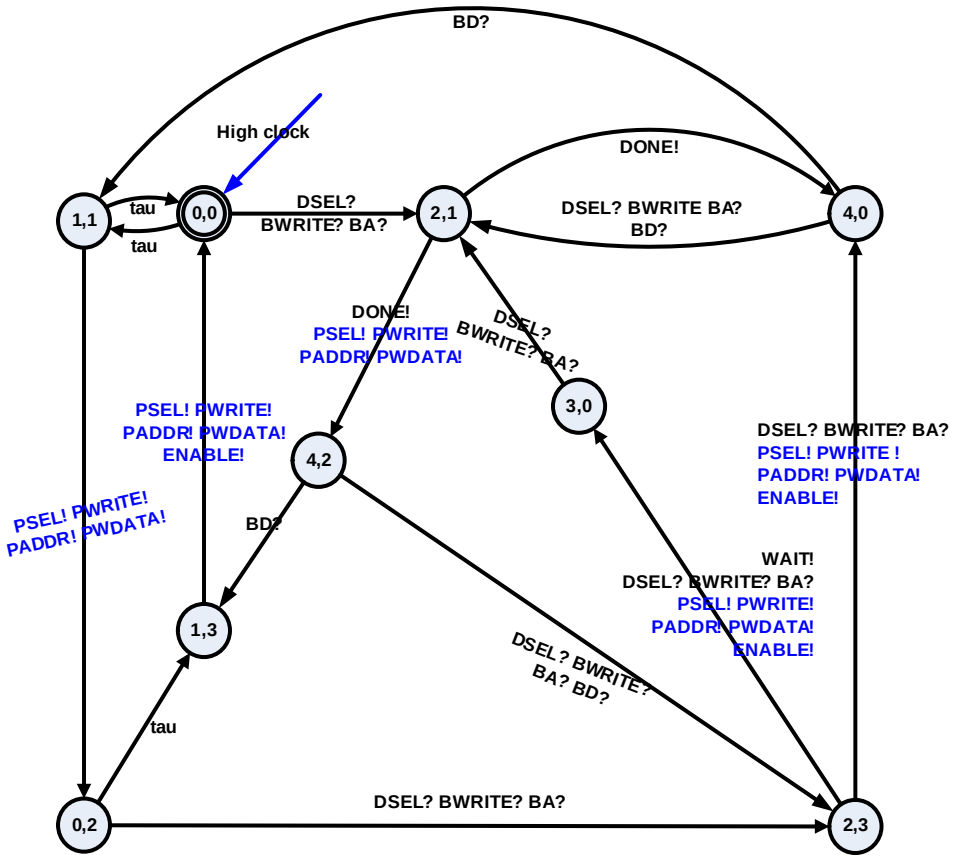

Figure 12: A suggested ASB to APB converter

5 Conclusions

In this report we have presented a general and comprehensive framework for modeling hardware protocols and for addressing the problem of protocol compatibility and protocol converter synthesis. The framework is the first to allow precise and detailed modeling of commercial protocols in a low abstraction level, and enables direct translation to HDL. We have presented a general definition for protocol compatibility and the protocol converter synthesis problem, formalize it and provide algorithms for automatic compatibility check. We have demonstrated the process of compatibility check and converter synthesis with commercial protocols AMBA ASB and AMBA APB, demonstrating that the framework is easily adaptable and practical for use with existing protocol specifications.

References

[1] Akella, J., and McMillan, K. L. Synthesizing converters between finite state protocols. In *ICCD* (1991), IEEE Computer Society, pp. 410–413.

[2] Alliane, V. S. I. http://www.vsi.org/.

[3] Androutsopoulos, V., Brookes, D., and Clarke, T. Protocol converter synthesis. *Computers and Digital Techniques, IEE Proceedings- 151*, 6 (2004), 391–401.

[4] Androutsopoulos, V., Clarke, T. J. W., and Brookes, M. Synthesis and optimization of interfaces between hardware modules with incompatible protocols. In *ISCAS (5)* (2003), pp. 613–616.

[5] Anjo, K., Okamura, A., and Motomura, M. Wrapper-based bus implementation techniques for performance improvement and cost reduction. *Solid-State Circuits, IEEE Journal of 39*, 5 (2004), 804–817.

[6] ARM. Advanced micro-controller bus architecture specification http://www.arm.com/products/solutions/ambahomepage.html.

[7] Borriello, G., and Katz, R. Synthesis and optimization of interface transducer logic. *Proceedings of the International Conference on Computer-Aided Design* (1987), 274–277.

[8] Choi, S., and Kang, S. Implementation of an on-chip bus bridge between heterogeneous buses with different clock frequencies. In *IWSOC* (2005), IEEE Computer Society, pp. 530–534.

[9] Cyr, G., Bois, G., and Aboulhamid, M. Generation of processor interface for soc using standard communication protocol. *Computers and Digital Techniques, IEE Proceedings- 151*, 5 (2004), 367–376.

[10] de Alfaro, L., and Henzinger, T. A. Interface automata. In *ESEC / SIGSOFT FSE* (2001), pp. 109–120.

[11] D'Silva, V., Ramesh, S., and Sowmya, A. Bridge over troubled wrappers: Automated interface synthesis. In *VLSI Design* (2004), IEEE Computer Society, pp. 189–194.

[12] D'Silva, V., Ramesh, S., and Sowmya, A. Synchronous protocol automata: A framework for modelling and verification of soc communication architectures. In *DATE* (2004), IEEE Computer Society, pp. 390–395.

[13] Gajski, D., Cho, H., and Abdi, S. General transducer architecture. Tech. Rep. TR 05-08, CECS Center for Embedded Computer Systems University of California, Irvine, August 2005.

[14] Hwang, Y., and Lin, S. Automatic protocol translation and template based interface synthesis for ip reuse in soc. *Circuits and Systems, 2004. Proceedings. The 2004 IEEE Asia-Pacific Conference on 1* (2004).

[15] IBM. A 32-, 64-, 128-bit core on-chip bus structure http://www-03.ibm.com/chips/products/coreconnect/.

[16] Ihmor, S., Loke, T., and Hardt, W. Synthesis of communication structures and protocols in distributed embedded systems. In *IEEE International Workshop on Rapid System Prototyping* (2005), IEEE Computer Society, pp. 3–9.

[17] Jou, J., Kuang, S., and Wu, K. A hierarchical interface design methodology and models for soc ipintegration. *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on 2* (2002).

[18] Narayan, S., and Gajski, D. Interfacing incompatible protocols using interface process generation. In *DAC* (1995), pp. 468–473.

[19] Partnership, O. C. P. I. Open core protocol specification http://www.ocpip.org.

[20] Passerone, R., de Alfaro, L., Henzinger, T. A., and Sangiovanni-Vincentelli, A. L. Convertibility verification and converter synthesis: two faces of the same coin. In *ICCAD* (2002), L. T. Pileggi and A. Kuehlmann, Eds., ACM, pp. 132–139.

[21] Passerone, R., Rowson, J. A., and Sangiovanni-Vincentelli, A. L. Automatic synthesis of interfaces between incompatible protocols. In *DAC* (1998), pp. 8–13.

[22] Roychoudhury, A., Thiagarajan, P. S., Tran, T.-A., and Zvereva, V. A. Automatic generation of protocol converters from scenario-based specifications. In *RTSS* (2004), IEEE Computer Society, pp. 447–458.

[23] Shin, D., and Gajski, D. Interface synthesis from protocol specification. Tech. Rep. CECS-02-13, Center for Embedded Computer Systems University of California, Irvine, dongwans,gajski@cecs.uci.edu, April 12, 2002.

[24] Smith, J., and Micheli, G. D. Automated composition of hardware components. In *DAC* (1998), pp. 14–19.

[25] Yun, C., Bae, Y., Cho, H., and Jhang, K. Automatic synthesis of interface circuits from simplified ip interface protocols. In *Asia-Pacific Computer Systems Architecture Conference* (2006), C. R. Jesshope and C. Egan, Eds., vol. 4186 of *Lecture Notes in Computer Science*, Springer, pp. 581–587.