Protocol Compatibility and Automatic 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
rameshari1958@gmail.com
GM India Science Lab
Bangalore India

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

UNSW-CSE-TR-0712
Technical Report, May 2007

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

Abstract

Hardware module reuse is common practice to deal with the increasing complexity of chip architectures and growing pressure to reduce time to market. In the absence of module interface standards, use of pre-designed modules in a "plug and play" fashion usually requires a converter between incompatible interface protocols. Though several approaches to such mediation have been proposed in the past, automation of protocol converter synthesis is yet to be realized. In this work, we present a state-machine based formalism for modeling bus based communication protocols, a notion of protocol compatibility and of protocol conversion. Using this formalism, we devise algorithms for checking protocol compatibility and for automatic converter synthesis. We report our experience with automatic converter synthesis for commercial bus protocols. The presented work is unique in its low abstraction level that enables precise modeling of protocol characteristics and simple translation to HDL.

Contents

1 Introduction **4**
- 1.1 Related Work . . . 4
- 1.2 Motivating Examples . . . 6

2 Formal Definitions **9**
- 2.1 Protocol Model . . . 9
- 2.2 Parallel Composition . . . 9
  - 2.2.1 An Example . . . 10
- 2.3 Paths . . . 11

3 Compatibility **13**
- 3.1 Defining Compatibility . . . 13
- 3.2 Checking Compatibility . . . 14
  - 3.2.1 $Paths(P, q_s, q_f)$ . . . 18
  - 3.2.2 An Example . . . 18

4 Converter Synthesis **21**
- 4.1 Defining the Converter Synthesis Problem . . . 21
- 4.2 Construction of a Correct Converter . . . 22
  - 4.2.1 Complete Parallel Composition . . . 22
  - 4.2.2 An Example . . . 23
  - 4.2.3 Inversion of Actions . . . 23
  - 4.2.4 An Inverted CPC Vs. a Correct Converter . . . 23
  - 4.2.5 Restricting $ICPC$ to Correct Behavior . . . 25
  - 4.2.6 Restricting Transitions . . . 25
  - 4.2.7 Restricting States . . . 26
  - 4.2.8 Finding the Largest Fixed Point . . . 29
  - 4.2.9 Examples . . . 30

5 Conclusions **33**

List of Figures

1 A typical SoC architecture 4
2 ASB slave timing diagram 7
3 APB slave timing diagram 8
4 Protocol models 8
5 The protocols. c_i is a control channel, a_i and d_i are data channels . 11
6 The parallel composition of the two protocols 11
7 System structure 13
8 The parallel composition of ASB and APB 16
9 The computation tree for ASB 19
10 System structure 21
11 The complete parallel composition of the protocols presented in Figure 5 24
12 A simple read operation 25
13 A simple write operation 26
14 No internal choice between outgoing transitions 26
15 Complete internal choice between outgoing transitions 32
16 The most general converter for P_1 and P_2 32
17 Minimized converter for P_1 and P_2 32

List of Algorithms

1 $ParallelComposition(P_1, P_2)$ 10
2 $Compatible(P_1, P_2)$ 14
3 $CheckTransitions(P = (Q, C, D, K, \rightarrow, q_s, q_f))$ 15
4 $CheckPaths(P = (Q, C, D, K, \rightarrow, q_s, q_f))$ 15
5 $CheckPath(\pi, type)$ 16
6 $CheckMerge(\pi)$ 17
7 $ComputeTree(P, initial\_state, final\_state)$ 18
8 $IsInPath(q, T)$ 19
9 $CompleteParallelComposition(P_1, P_2)$ 23
10 $Inverse\_Tau(q)$ 24
11 bool $inclusion(S_1, S_2)$ 28
12 $StateLabel(q)$ 29
13 $Restrict(C, B)$ 30

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 may 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.

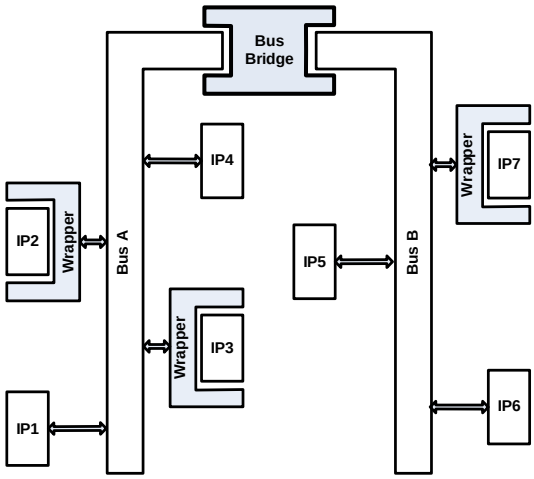

Figure 1: A typical SoC architecture

Much research has been dedicated to "the converter synthesis" problem of SoC communication, from attempts to standardize interface protocols [1,2,7,17], through methodologies for reusable design of IPs [19], research on the design of glue logic, some for specific protocols [6, 9, 10], and some of a more general approach [12–15, 18, 21–24]. Attempts have been made to automate the process of generation of glue logic at different levels [3–5, 13, 14, 21–23].

Different approaches and models have been investigated for automation of converter synthesis: Timing diagrams [8], data queuing [15,24], message sequencing [23], FSM based protocol modeling [3–5, 13, 14, 22] 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 and of FSM based models.

In early work [8] 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 [3] 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 [4, 5, 14] and is the foundation of our work.

Later work [20] 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 [25], 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 [15, 24] 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 and 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 [22]. Their technique cannot be easily extended to different kinds of data and clock speed mismatches. In later work, Passerone et al. [21] stated that the above methods lack a mathematically sound foundation 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 [13,14]. The protocols in this work are represented as two FSMs, the outcome of the algorithm is a third FSM to be used as the wrapper for either of the protocols. Methods for dealing with mismatched data types and clock periods are also presented though they are not integrated into the given algorithm.

Another approach presented in [23], 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 [4,5] 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 [16] seem to have adopted the approach presented in [13, 14] and in [16] they demonstrate its implementation over a great simplification of AMBA AHB and VCI BVCI.

Recent work [26] also relies on [13,14] as well and shows how working at a higher level of abstraction reduces the size of models and thereby 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 the 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 any 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 a notion of compatibility and of protocol conversion, and propose algorithms for checking compatibility and for automatic synthesis of protocol converters

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 APB is given in Figure 3 (both taken from [7]).

Other than the different Signal naming between the two protocols, ASB is a much more complicated protocol than APB 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 [13]. The notation of the models is explained in details in section 2.1 but the incompatibility of the two models can be seen easily.

In APB, 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 responsds 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 [7])

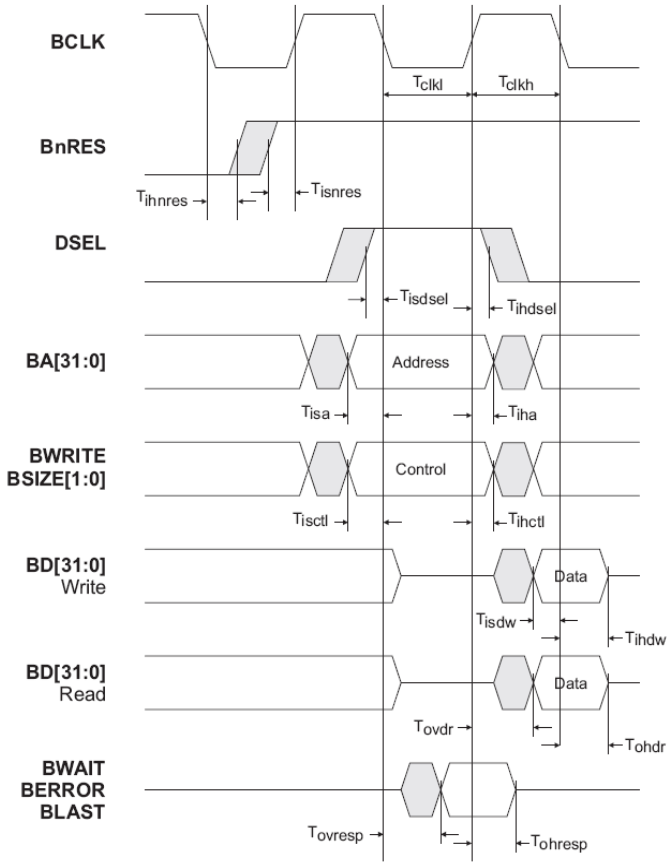

Figure 2: ASB slave timing diagram

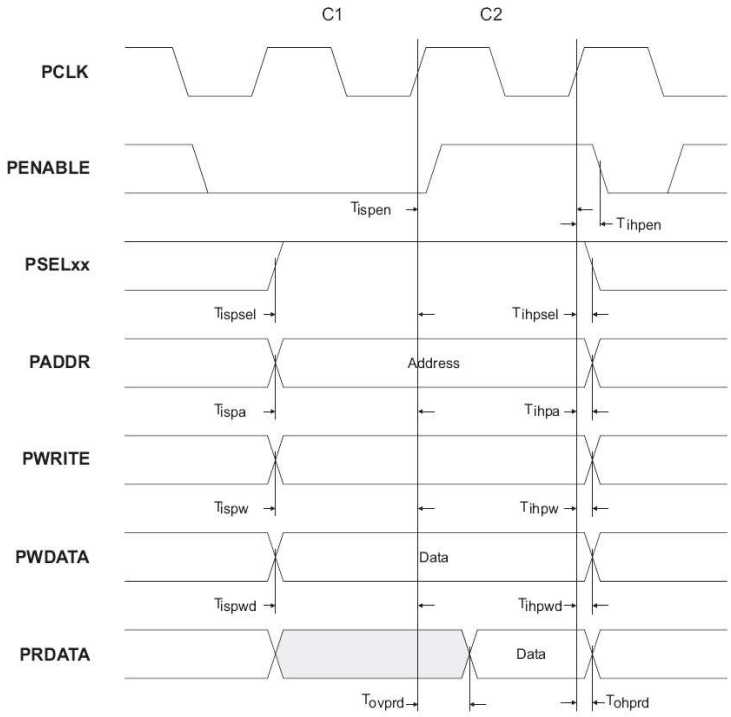

Figure 3: APB slave timing diagram

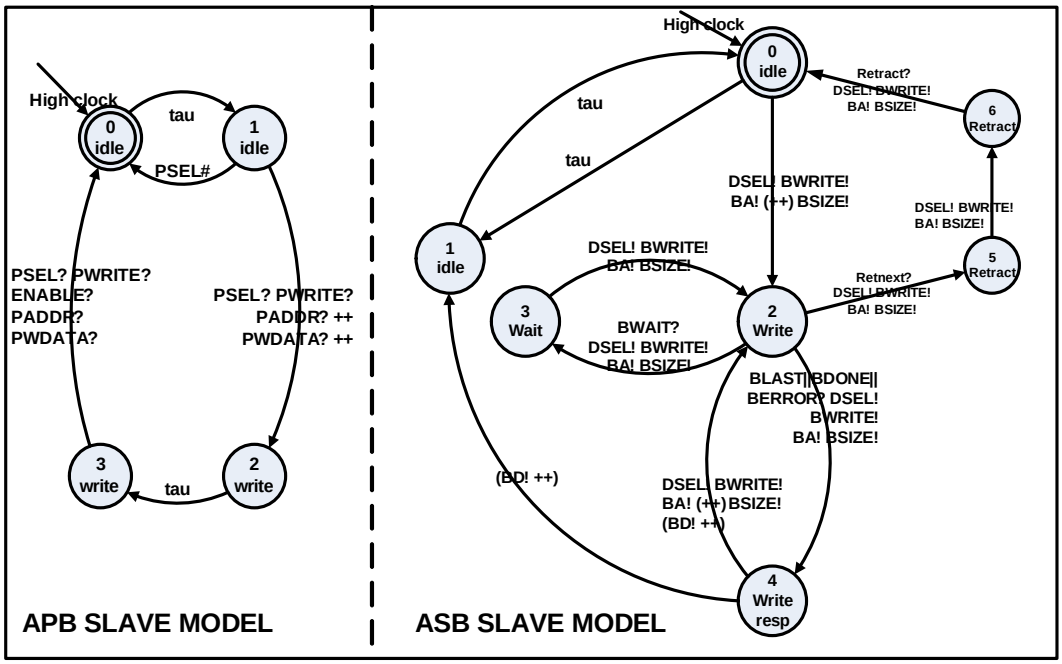

Figure 4: Protocol models

2 Formal Definitions

2.1 Protocol Model

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 signal value on a control channel, denoted $c?$ and $c\#$ respectively and write to a control channel, denoted $c!$. A protocol can also read values from or write values to a data channel d, denoted $d?$ 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 (i.e. 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 action 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 a natural number. A protocol performs channel and counter actions.

In order to guarantee that counter values are bounded, it is a requirement of the protocols to have reset actions infinitely often. In particular, a reset should be used for any data transfers in bursts of unbounded length. In these cases the counter value does not represent the actual amount of data transfered but allows new data items to be distinguied from data repetition items and data repetitions.

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, 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 power set of A_P).*

2.2 Parallel Composition

We are interested in the behavior of protocols executing concurrently, which 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 sets of 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* $c! \notin S_1$.

$\texttt{may}(S_1, S_2)$ is true for two sets of actions S_1 and S_2 when the two sets satisfy each others controls requirements.

Definition 3 (Parallel Composition (PC)) *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,* $P_1 \| P_2 = (Q_1 \times Q_2, (C_1 \cup C_2), (D_1 \cup D_2), K_1 \cup K_2, \rightarrow, (q1_s, q2_s), (q1_f, q2_f))$, *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, such that* $\texttt{may}(S_1, S_2)$ *is true and* S *is the set of actions occurring in* $S_1 \cup S_2$.

For two protocols that do not share a channel naming convention, channel mapping information is needed in order to compute the parallel composition.

An Algorithm for the construction of a parallel composition is given in Algorithm 1. This algorithm does not compute the entire PC but only the states and transitions reachable from the initial state $(q1_s, q2_s)$.

Algorithm 1 *ParallelComposition*(P_1, P_2)

Input: Two Protocols

Output: $PC = P_1 \| P_2$ the Parallel Composition of the two protocols, including only the states reachable from the initial state

```
1:  Q_PC = (q1_s, q2_s);      // The list of states of PC, containing the initial state
2:  for all states (q1, q2) ∈ Q_PC do
3:    for all transitions q1 --S1-->_1 q1' ∈ P_1 do
4:      for all transitions q2 --S2-->_2 q2' ∈ P_2 do
5:        if may(S_1, S_2) then
6:          Add transitions (q1, q2) --S1∪S2--> (q1', q2') to PC
7:          if (q1', q2') ∉ PC then
8:            Add state (q1', q2') to Q_PC
9:          end if;
10:       end if;     // may
11:     end for;      // All transitions in P_2
12:   end for;      // All transitions in P_1
13: end for;      // All states in PC
```

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.

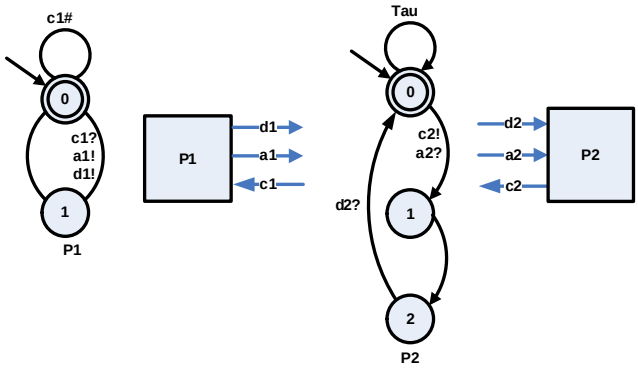

Figure 5: The protocols. c_i is a control channel, a_i and d_i are data channels

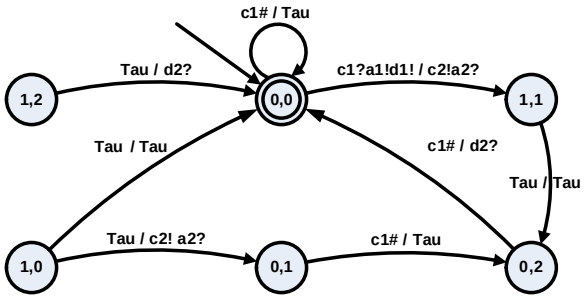

Figure 6: The parallel composition of the two protocols

2.3 Paths

A path in a protocol P is a sequence of alternating 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, also referred to as the length of π.
- For a path π in a finite state machine $P_1 \| P_2$,

  $$\pi = (q1_0, q2_0) \xrightarrow{S_1} (q1_1, q2_1) \ldots \xrightarrow{S_k} (q1_k, q2_k)$$

  such that $qi_j, \in P_i$, there exist matching paths π_1 and π_2 in P_1 and P_2.

  $$\pi_1 = q1_0 \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_k} q1_k$$

$$\pi_2 = q2_0 \xrightarrow{S2_1} q2_1 \ldots \xrightarrow{S2_k} q2_k$$

such that $S1_i \cup S2_i = S_i$ (implied by the definition of parallel composition). Let $Projection(\pi, P_i)$ denote the path in protocol P_i matching π.

($\pi_1 = Projection(\pi, P_1)$ and $\pi_2 = Projection(\pi, P_2)$)

- A *cycle* is a path such that $q_0 = q_k$.

3 Compatibility

3.1 Defining 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:

1. Data is read by one protocol only when written by the other.
2. A given data item is read exactly once.
3. No undefined behaviors can be reached (at every state at least one transition should be enabled), 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 7. We define compatibility in terms of constraints over paths in the parallel composition of two protocols.

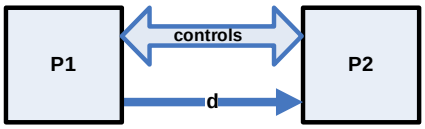

Figure 7: System structure

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 $\pi_1 = Projection(\pi, P_1) = q1_s \xrightarrow{S_1^1} q1_1 \ldots \xrightarrow{S_k^1} q1_f$ and $\pi_2 = Projection(\pi, P_1) = q2_s \xrightarrow{S_1^2} q2_1 \ldots \xrightarrow{S_k^2} q2_f$ 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 index 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, guaranteeing 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) This 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 several distinct data items if read multiple times. Requirement 3 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 the general case where there is more than one data channel, condition 2 should hold for every channel independently.

3.2 Checking Compatibility

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

Algorithm 2 $Compatible(P_1, P_2)$

Input: Two protocols P_1 and P_2.
Output: TRUE if the protocols are found to be compatible, FALSE otherwise.

```
1: PC = ParallelComposition(P1, P2);     // see Algorithm 1
2: if CheckTransitions(PC) == FALSE then
3:    return FALSE;     // invalid data is read
4: end if;
5: if (states of PC) ≠ (states of Direct_Paths(PC)) ∪ (states of Cycles(PC)) then
6:    return FALSE;     // PC has deadlocks - states that cannot reach the final
      state
7: end if;
8: if CheckPaths(PC) == FALSE; then
9:    return FALSE;
10: end if;
11: return TRUE;     // the protocols are compatible
```

In Algorithm 2 Line 2 refers to Algorithm 3 where all transitions are checked individually to guarantee that only valid data is read (i.e. data is read by one protocol only when written by the other) corresponding to requirement 2a in Definition 4.

In line 5 (Algorithm 2) the parallel composition of the two protocols is checked. This check will fail if the reachable component of the parallel composition (from the initial state) contain deadlocks - states that cannot reach the final state, corresponding to requirement 3 in Definition 4.

The $CheckPaths(PC)$ Algorithm (Algorithm 4) called in line 8 of $Compatible(P_1, P_2)$, corresponds to requirements 2b and 2c in Definition 4 and performs several checks:

1. Lines 1 to 5 in Algorithm 4: Checking that all direct paths alternate correctly between new read and write actions.

2. Lines 6 to 10 in Algorithm 4: Checking that all cycles alternate correctly between new read and write actions.

3. Lines 11 to 15 in Algorithm 4: Checking that all cycles merge correctly in terms of alternating new read and write actions.

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

```
1: for all t ∈→ do
2:   for all d ∈ D do
3:     if "d?" ∈ t and "d!" ∉ t then
4:       return FALSE;
5:     end if;
6:   end for;
7: end for;
8: return TRUE;     // all transitions have passed the check
```

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

Input: P: a parallel composition of protocols

```
1: for all π ∈ Direct_Paths(P) do
2:   if CheckPath(π, DIRECT) == FALSE then
3:     return FALSE;
4:   end if;
5: end for;
6: for all π ∈ Cycles(P) do
7:   if CheckPath(π, CYCLE) == FALSE then
8:     return FALSE;
9:   end if;
10: end for;
11: for all π ∈ Cycles(P) do
12:   if CheckMerge(π) == FALSE then
13:     return FALSE;
14:   end if;
15: end for;
16: return TRUE;
```

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.

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 8. This would pass the transitions check but will fail the check that all reachable states can reach the final state, as in line 5 in Algorithm 2 (state $(2, 1)$ can be reached from the initial state but does not have a path to the the final state)

Algorithm 5 $CheckPath(\pi, type)$

Input: a single path π of type DIRECT or CYCLE.
Output: TURE if the path passes the check, FALSE otherwise.

1: $\pi_{P1} = Projection(\pi, P_1)$
2: $\pi_{P2} = Projection(\pi, P_2)$
3: **for all** $d \in D$ **do**
// for every data channel in the parallel composition
// assuming d is input to P_2
4: $Writes = New(\pi_{P_1}, d) = \{i_1 < i_2 \cdots < i_{n1}\}$;
5: $Reads = New(\pi_{P_2}, d) = \{j_1 < j_2 \cdots < j_{n2}\}$;
6: **if** d is input to P_1 **then**
7: swap($Writes, Reads$)
8: **end if**;
9: **if** $(type == CYCLE) \wedge (j_1 < i_1)$ **then**
10: swap($Writes, Reads$) // a cycle can begin in a read
11: **end if**;
12: **if** $n1 \neq n2$ **then**
13: **return** FALSE; // number of writes does not match number of reads
14: **else**
15: $j_0 = 0$;
16: $i_{n1+1} = k + 1$;
17: **for all** ℓ such that $1 \leq \ell \leq n1$ **do**
18: **if** $(j_{\ell-1} \geq i_\ell)$ or $(i_\ell > j_\ell)$ or $(j_\ell \geq i_{\ell+1})$ **then**
19: **return** FALSE
20: **end if**;
21: **end for**; // all indices 1 to $n1$
22: **end if**; // the same number of reads and writes
23: **end for**; // for all data channels
24: **return** TRUE;

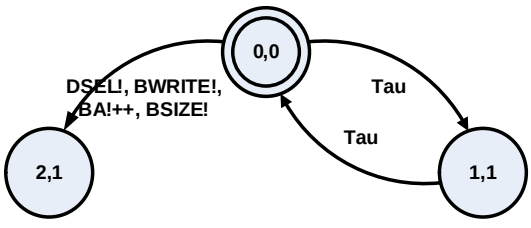

Figure 8: The parallel composition of ASB and APB

Algorithm 6 $CheckMerge(\pi)$

Input: a cycle of the parallel composition P, of the form $\pi = q_0 \xrightarrow{S_1} q_1 \ldots \xrightarrow{S_k} q_k$
Output: TRUE if all cycle merges pass the check, FALSE otherwise.

1: $Writes_\pi = New(\pi_{P_1}, d) = \{i_1 < i_2 \cdots < i_n\}$;
2: $Reads_\pi = New(\pi_{P_2}, d) = \{j_1 < j_2 \cdots < j_n\}$;
3: **for all** $\pi_1 \in Direct\_Paths(P) \cup Cycles(P) \neq \pi$ **do**
// $(\pi_1 = q1_0 \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_\ell} q1_\ell)$
4: $Writes_{\pi 1} = New(\pi 1_{P_1}, d) = \{i1_1 < i1_2 \cdots < i1_{n1}\}$;
5: $Reads_{\pi 1} = New(\pi 1_{P_2}, d) = \{j1_1 < j1_2 \cdots < j1_{n1}\}$;
6: **for** m from 1 to $(\ell - 1)$ **do**
7: **if** $q_0 == q1_m$ **then**
// state number m is the merging point
8: find x s.t. $(i1_x < m) \wedge (i1_{x+1} \geq m)$
// x is the transition index of the last write before the merge
9: find y s.t. $(j1_y < m) \wedge (j1_{y+1} \geq m)$
// y is the transition index of the last read before the merge
10: **if** $(i_1 \leq j_1) \wedge (x > y)$ **then**
// the merge results in a written data item never read
11: **return** FALSE;
12: **end if**;
13: **if** $(i_1 > j_1) \wedge (x \leq y)$ **then**
// the merge results in a written data item read twice
14: **return** FALSE;
15: **end if**;
16: **end if**; // $q_0 == q1_m$
17: **end for**; // i from 1 to $(\ell - 1)$
18: **end for**; // all direct paths and cycles
19: **return** TRUE;

Algorithm 4 uses two lists, *Direct_Paths* and *Cycles*. These two lists refer to paths in the parallel composition of the two protocols and they essentially break the possibly infinite list of all paths from initial to final state of the parallel composition into atomic elements, made of two finite states.

The following section (3.2.1) elaborates on the extraction of the two lists for a general graph.

3.2.1 $Paths(P, q_s, q_f)$

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.

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 the 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.

An algorithm for the construction of the tree is given in Algorithm 7.

Algorithm 7 *ComputeTree*(P, *initial_state*, $final\_state$)

```
1: i = 0;     // Levels counter
2: Level[0] = initial_state;
3: while Level[i] ≠ ∅ do
4:    for all states q ∈ Level[i] do
5:       if (i == 0) or ((q is not in path) and (q ≠ final_state)) then
  //          (see Algorithm 8 for finding whether q is in the path)
6:          for all outgoing transitions q --S--> q' do
7:             add q' to Level[i + 1];
8:             add q --S--> q' to the tree;
9:          end for;
10:      end if;
11:   end for;
12:   i++;
13: end while;
```

3.2.2 An Example

The computation tree of the ASB model from Figure 4 is presented in Figure 9

Algorithm 8 $IsInPath(q, T)$

```
1: if q == T.root then
2:    return TRUE
3: else
4:    to_check = q;
5:    while to_check ≠ T.root do
6:       to_check = parent(to_check);
7:       if to_check == q then
8:          return TRUE;
9:       end if;
10:   end while;
11:   return FALSE
12: end if;
```

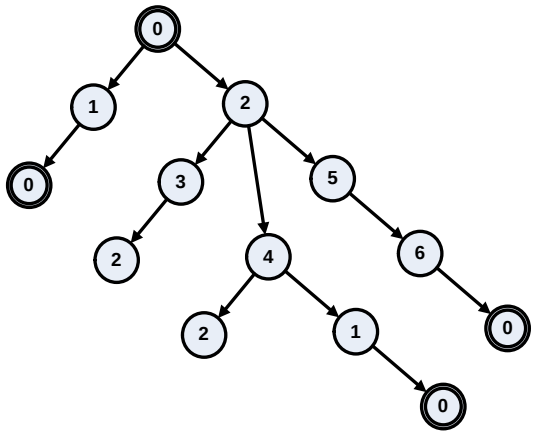

Figure 9: The computation tree for ASB

$$DirectPaths = \{ \quad \pi_1 = 0 \xrightarrow{Tau} 1 \xrightarrow{Tau} 0, \qquad Cycles = \{ \quad \pi_4 = 2 \xrightarrow{S_7} 3 \xrightarrow{S_8} 2,$$
$$\pi_2 = 0 \xrightarrow{S_1} 2 \xrightarrow{S_2} 4 \xrightarrow{S_3} 1 \xrightarrow{Tau} 0, \qquad \pi_5 = 2 \xrightarrow{S_2} 4 \xrightarrow{S_9} 2\}$$
$$\pi_3 = 0 \xrightarrow{S_1} 2 \xrightarrow{S_4} 5 \xrightarrow{S_5} 6 \xrightarrow{S_6} 0 \ \}$$

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! (++)",

4 Converter Synthesis

4.1 Defining the Converter Synthesis Problem

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:

1. Data is read by one protocol (/the converter) only when written by the converter (/a protocol).
2. A given data item is read exactly once.
3. Every data item written by $P_1(/P_2)$ to the converter will be written by the converter to P_2 $(/P_1)$.
4. 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)
5. No undefined behaviors can be reached, and every transaction can terminate within finite time (equivalent to a finite number of transitions).

In most cases, where the two protocols are not synchronized on read and write actions over data channels, a converter needs to temporarily store data received from one protocol before writing it to the other. We assume a buffer of finite depth and that buffer state information is available at every state.

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. We also assume a finite buffer of depth B for storage of data items.

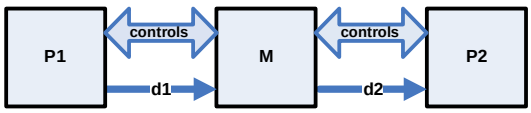

Figure 10: System structure

Definition 5 (Converter Synthesis Problem) *Given two incompatible protocols P_1 and P_2, synthesize a finite state machine M with bounded counters 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 index ℓ it holds that $i_\ell \leq j_\ell \leq i_{\ell+B}$ where $1 \leq \ell \leq n$ and B is the size of the buffer dedicated to channel $d1$ (and $d2$).

where $q(m)_s$ is a state at which both protocols are in their initial states and all buffers are empty, and qm_f is a state at which both protocols are in their finite states and all buffers are empty.

The first two requirements guarantee that data items are read exactly once, only when written and no undefined behaviors can occur (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 terms of data storage, it reflects the need to avoid buffer underflow (reading a data item from an empty buffer) and overflow (writing a data item to a full buffer), both resulting in data corruption. In the general case where there are more data channels, constraint 3 of definition 5 should hold for every pair of mapped channels independently.

4.2 Construction of a Correct Converter

We present here an algorithm for the construction of a correct converter. This Algorithm takes an input of two protocols and data channel mapping information and returns the most general correct converter, out of which smaller and more deterministic converters can be extracted. The algorithm includes the following stages: (1) construction of a complete parallel composition as described in 4.2.1, (2) inversion of actions (section 4.2.3), and (3) restriction to correct behaviour (section 4.2.5).

4.2.1 Complete Parallel Composition

The construction of the converter begins with the cross product of the two protocols defined here as the complete parallel composition, or *CPC*:

Definition 6 (Complete Parallel Composition (*CPC*)) *The complete parallel composition $P_1 \|_C P_2$ 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 $P_1 \|_C P_2 = (Q_1 \times Q_2, (C_1 \cup C_2), (D_1 \cup D_2), K_1 \cup K_2, \rightarrow, (q1_s, q2_s), (q1_f, q2_f))$ where $(q1, q2) \xrightarrow{S} (q1', q2')$ is a transition of $P_1 \|_C P_2$ iff $q1 \xrightarrow{S_1}_1 q1'$ and $q2 \xrightarrow{S_2}_2 q2'$ are transitions of P_1 and P_2 respectively and S is the set of actions occurring in $S_1 \cup S_2$ including complementary actions.*

Notice that the complete parallel composition (*CPC*) differs from the parallel composition (*PC*) as in definition 3, in the removal of the $\texttt{may}(S_1, S_2)$ requirement from the transitions, and therefore *CPC* includes all possible pairs of transitions and describes the most general behavior of the two protocols in parallel. (whereas

in PC we allow only for control compatible transitions - that satisfy the control input requirements of both protocols to occur in parallel).

An algorithm for the construction of a complete parallel composition is given in Algorithm 9. Under the assumption that the protocols $(P1, P2)$ contain only states that can be reached from their initial states, This algorithm computes the entire CPC that by definition does not contain any states that are not reachable from the initial state $(q1_s, q2_s)$.

Algorithm 9 *CompleteParallelComposition*(P_1, P_2)

Input: Two Protocols

Output: $CPC = P_1 \|_C P_2$ the Complete Parallel Composition of the two protocols.

1: $Q_{CPC} = (q1_s, q2_s)$; // States of CPC, initialized with the initial state
2: **for all** states $(q1, q2) \in Q_{CPC}$ **do**
3: **for all** transitions $q1 \xrightarrow{S_1}_1 q1' \in P_1$ **do**
4: **for all** transitions $q2 \xrightarrow{S_2}_2 q2' \in P_2$ **do**
5: Add transitions $(q1, q2) \xrightarrow{S_1 \cup S_2} (q1', q2')$ to CPC
6: **if** $(q1', q2') \notin CPC$ **then**
7: Add state $(q1', q2')$ to Q_{CPC}
8: **end if**;
9: **end for**; // All transitions in P_2
10: **end for**; // All transitions in P_1
11: **end for**; // All states in CPC

4.2.2 An Example

Considering the two protocols P_1 and P_2 as presented in Figure 5, the complete parallel composition of the two protocols according to definition 6 is illustrated in Figure 11.

4.2.3 Inversion of Actions

Considering that all outputs of the protocols need to be used as inputs to the converter and vice-versa, in order to use the CPC for the construction of the converter we would like to invert all actions of the CPC by replacing every action with its complementary action.

Complementary actions can be seen in Table 1, where $Tau(c)$ means that the inverted label should not include any output action on channel c, and C^O is the set of output control channels of the protocol (input to the converter).

A minimized complementary action for Tau, for transitions outgoing of a state q in a protocol can be calculated using Algorithm 10, as only control output channels that are referred by other outgoing transitions in the protocol need to be addressed.

4.2.4 An Inverted CPC Vs. a Correct Converter

Following definition 5 for a correct converter, it is easy to see that an inverted CPC ($ICPC$) is compatible with both protocols (every path in each protocol is the

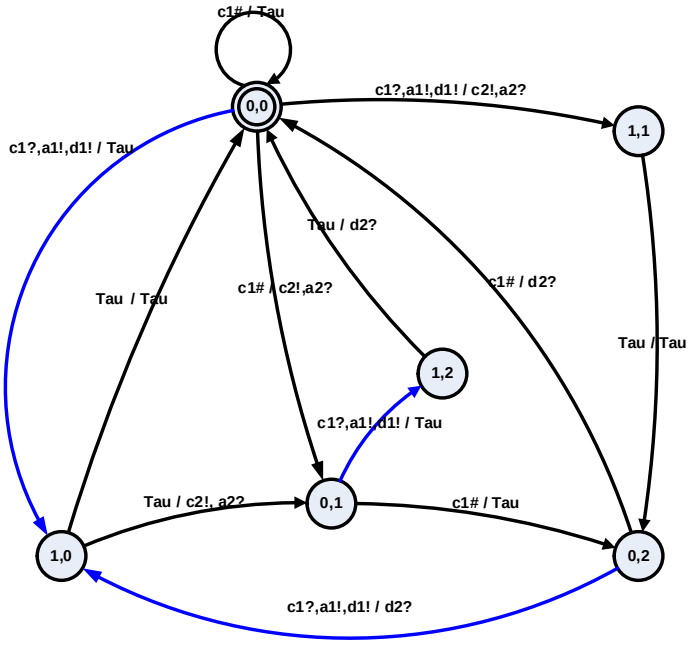

Figure 11: The complete parallel composition of the protocols presented in Figure 5

Algorithm 10 $Inverse\_Tau(q)$

Input: a state q.

Output: a set of actions over control channels, $L \in \mathcal{P}(A_C)$.

1: $L = \emptyset$; // initializing L
2: **for all** outgoing transitions $q \xrightarrow{S} q'$ **do**
3: **for all** control actions $c \in C$ **do**
4: **if** $c! \in S$ **then**
5: $L = L \cup (c\#)$;
6: **end if**;
7: **end for**;
8: **end for**;
9: **return** L;

Table 1: Complementary actions

Channel type	Action	Complementary action
Data	$d?$	$d!$
	$d!$	$d?$
Control	$c?$	$c!$
	$c!$	$c?$
	$c\#$	$Tau(c)$
Counter	$reset(k)$	$reset(k)$
	$k++$	$k++$
	$k = v$	$k = v$
General	Tau	$c\#, \forall c \in C^O$

inverted projection of a path in the $ICPC$), satisfying requirements 1 and 2. All that is left to do is to constrain the $ICPC$ to comply with requirement 3 and we are left with a correct converter.

4.2.5 Restricting $ICPC$ to Correct Behavior

Since the $ICPC$ includes all possible parallel transitions, we can turn it into a converter by forcing the converter to avoid transitions that may lead to incorrect behavior while keeping the converter compatible with both protocols. By incorrect behavior we refer to any behavior that violates definition 5, section 3. Since this condition deals with a need to avoid buffer overflow/underflow, we add a condition to each transition referring to the buffer state (the amount of data items in the buffer) to restrict transitions that lead to an overflow/underflow.

4.2.6 Restricting Transitions

A simple read operation between two states in a converter, as illustrated in Figure 12, might cause an overflow of the buffer if the transition is taken when the buffer is already full. Therefore, to avoid the possible overflow we add a condition to the transition that restricts taking the transition if the buffer is full (B represents the depth of the buffer)

Figure 12: A simple read operation

In the same manner, a simple write operation between two states, as illustrated in Figure 13, might cause an underflow of the buffer if the transition is taken when the buffer is empty. Therefore, to avoid the possible underflow we add a condition to the transition that restricts taking the transition if the buffer is empty.

The restrictions added to the transitions are made under the assumption that the destination state (state labeled 1) can be reached at any state of the buffer ($0 \leq Data\_buffer \leq B$), which may not be true.

Figure 13: A simple write operation

In the more general case where a destination state q should only be reached for a specific range of values between x and y ($0 \leq x \leq Data\_buffer \leq y \leq B$), the restriction on any incoming transition should never lead to a breach of the destination state allowed range, and so for a single write operation on a transition incoming state q the condition should be $min(0, x-1) \leq Data\_buffer < y$, while for a single read operation on a transition incoming state q the condition should be $x < Data\_buffer \leq max(B, y+1)$.

The only type of actions that impose restrictions on a transition are reads and writes of new data items (where there is an increment or a reset of the data counter), and only when there is a difference between the amount of data items written and read.

4.2.7 Restricting States

Once a condition is imposed on a transition, it might affect the state of the buffer at which the source state (state 0 in the examples) should be reached. Whether the condition affects the source state or not depends on the other transitions outgoing from the same states and the relation between their incoming controls. In other words, the allowed range of buffer levels in a state, depends on the level of internal choice at the state.

For example, in a source state with no internal choice, where the outgoing transition is chosen by input controls alone (as illustrated in Figure 14) the state should be restricted to the intersection of all buffer level conditions in all outgoing transitions, and so in the example illustrated in Figure 14 the allowed range of buffer levels in state $q1$ is $3 < Data\_buffer < B-2$. While in a source state with full internal choice (meaning that the choice is independent of control inputs, as illustrated in Figure 15), the state should be restricted to the conjunction of all buffer level conditions in all outgoing transitions, and so in the example illustrated in Figure 15 the allowed range of buffer levels in state $q1$ is $0 < Data\_buffer \leq B$.

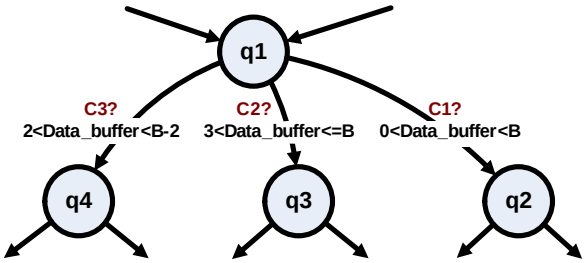

Figure 14: No internal choice between outgoing transitions

We use Definition 7 to distinguish between different levels of choice between transitions.

Definition 7 (`inclusion`) *label A includes label B (denoted $B \subseteq A$) iff label A accepts all input control values that are accepted by label B.*

An algorithm for $B \subseteq A$ is given in Algorithm 11.

A complete choice between transitions Labeled L_1 and L_2, is possible only when $L_1 = L_2$, that is, $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$. On the other hand, no choice is possible when $L_1 \nsubseteq L_2$ and $L_2 \nsubseteq L_1$.

For example, Table 2 details the values of controlls accepted by $L_3 = c_1?, c_2?$ and $L_4 = c_1?, c_3?$:

c_1	c_2	c_3	$L_3 = c_1?, c_2?$	$L_4 = c_1?, c_3?$
0	–	–	$FALSE$	$FALSE$
1	0	0	$FALSE$	$FALSE$
1	0	1	$FALSE$	$TRUE$
1	1	0	$TRUE$	$FALSE$
1	1	1	$TRUE$	$TRUE$

Table 2: Input Control Values

Clearly, no relation of inclusion exists between L_3 and L_4 and therefore we can conclude that there is no choice between the two transitions. A source state of such two transitions should be restricted to the conjunction of the transitions conditions, since it is not clear which transitions will be chosen once we reach the source state.

In a different case where there is inclusion in one direction only - $L_5 \subseteq L_6$ but $L_6 \nsubseteq L_5$ there is some level of choice. For example, Table 3 details the values of controlls accepted by $L_5 = c_1?, c_2?$ and $L_6 = c_1?$:

c_1	c_2	$L_5 = c_1?, c_2?$	$L_6 = c_1?$
0	–	$FALSE$	$FALSE$
1	0	$FALSE$	$TRUE$
1	1	$TRUE$	$TRUE$

Table 3: Input Control Values

L_5 is included in L_6 ($L_5 \subseteq L_6$), since for every input where L_5 is $TRUE$, L_6 is also $TRUE$. In this case we say that there is a limited choice since the converter can be forced to take L_6 but cannot be forced to take L_5. For an input of $c_1!, c_2!$ the converter can choose between the two transitions. In this case, a source state of such two transitions should be restricted to the condition of L_6 as a cover for a worst case scenario.

An algorithm for computing the state buffer condition with respect to its outgoing transitions is given in Algorithm 12. The Algorithm keeps two lists of label information: *Controls_List* holds lists of input control sets that have been found in outgoing transitions and have no inclusion relations between different entries in the list, and *Conditions_List* whose items hold the buffer state conditions corresponding to the items in *Controls_List*.

Starting with empty lists it loops on all outgoing transitions (lines 3 to 15), and examine the relation between the current observed transition. If the new label

Algorithm 11 bool $inclusion(S_1, S_2)$

Input: Transitions labels.

Output: TRUE if label S_1 is included in label S_2, FALSE otherwise

1: **for all** input action $A \in S_1$ over boolean control channel c **do**
2: **if** $\neg(A \in S_2 \vee c \notin S_2)$ **then**
3: **return** FALSE;
4: **end if**
5: **end for**;
6: **for all** input action $A \in S_1$ over non-boolean control channel c **do**
7: **if** $(c?v_1 \in S_1) \wedge \neg((c?v_1 \in S_2) \vee (c?v_2 \notin S_2 \text{ s.t. } v_1 \neq v_2) \vee (c \neq v_2 \in S_2 \text{ s.t. } v_1 \neq v_2) \vee (c \notin S_2))$ **then**
8: **return** FALSE;
9: **else if** $(c \neq v_1 \in S_1) \wedge \neg((c \neq v_1 \in S_2) \vee (c \notin S_2))$ **then**
10: **return** FALSE;
11: **end if**
12: **end for**;
13: **for all** data counter $k \in S_1$ **do**
14: **if** $k < x_1 \in S_1$ **then**
15: **if** $\neg((k < x_2 \in S_2 \text{ s.t } x_1 \leq x_2) \vee (k \neq x_2 \in S_2 \text{ s.t } x_1 \leq x_2) \vee (k \notin S_2))$ **then**
16: **return** FALSE;
17: **end if**;
18: **else if** $k > x_1 \in S_1$ **then**
19: **if** $\neg((k > x_2 \in S_2 \text{ s.t } x_1 \geq x_2) \vee (k \neq x_2 \in S_2 \text{ s.t } x_1 \geq x_2) \vee (k \notin S_2))$ **then**
20: **return** FALSE;
21: **end if**;
22: **else if** $k = x_1 \in S_1$ **then**
23: **if** $\neg((k = x_1 \in S_2) \vee (k \neq x_2 \in S_2 \text{ s.t. } x_1 \neq x_2) \vee (k > x_2 \in S_2 \text{ s.t } x_1 \geq x_2) \vee (k < x_2 \in S_2 \text{ s.t } x_1 \leq x_2) \vee (k \notin S_2))$ **then**
24: **return** FALSE;
25: **end if**;
26: **else if** $k \neq x_1 \in S_1$ **then**
27: **if** $\neg((k \neq x_1 \in S_2) \vee (k \notin S_2))$ **then**
28: **return** FALSE;
29: **end if**;
30: **end if**;
31: **end for**
32: **return** TRUE;

control inputs are the same as an entry i in $Control\_List$ (line 4), the conjunction of the conditions is taken as $Conditions\_List[i]$. If the new label control inputs include entry i in $Controls\_List$ (line 6), the values for entry i in both lists are replaced with the values found in the observed transition. If the new label control inputs are included in an item in $Control\_List$ (line 9), it is ignored. If the new label control inputs are not included in any item in $Controls\_List$(line 11), the new label is taken as a new entry to $Controls\_List$ and the buffer state condition of the transition is its corresponding value in the $Conditions\_List$.

Once all outgoing transitions have been examined, we are left with a list of conditions that correspond to input values that have no inclusion relations between them and so the state condition is computed as the disjunction of all entries in $Conditions\_List$ (lines 16 to 19).

Algorithm 12 $StateLabel(q)$

Input: A state q.

Output: A condition on the number of data items in the buffer when entering the state with respect to the conditions of the outgoing transitions from state q.

```
1:  Conditions_List = ∅;
2:  Controls_List = ∅;
3:  for all labels (L) of outgoing transitions of q do
4:     if ∃i s.t Controls_List[i] == control[L] then
   //        input is the same as an entry already in the list
5:        Conditions_List[i] = Conditions_List[i] ∨ condition[L];
6:     else if ∃i s.t. Controls_List[i] ⊆ control[L] then
   //        an item in the list is included in the current explored label
7:        Controls_List[i] = control[L];
8:        Conditions_List[i] = condition[L];
9:     else if ∃i s.t. control[L] ⊆ Controls_List[i] then
   //        the current explored item is included in an entry in the list
10:       continue;
11:    else
   //        no inclusions, create new entries in the lists
12:       add control[L] to the list Controls_List;
13:       add condition[L] to the list Conditions_List;
14:    end if;
15: end for;     // for all outgoing transitions
16: Condition = NULL;
17: for all items (L) ∈ Conditions_List do
18:    Condition = Condition ∧ L;
19: end for;
20: return  Condition;
```

4.2.8 Finding the Largest Fixed Point

The function of updating the range of allowed buffer status for a specific state, is monotonic

over a closed set - the range of allowed buffer values is bounded to the size of the buffer,

and when applied iteratively, returns ranges bounded between an empty set and the range of the previous iteration. These characteristics of the function guarantee its convergence [11], and with the initial range of $[0..B]$ we are guaranteed to converge to the largest fixed point - the widest range of buffer status values allowed at a specific state.

Algorithm 13 takes as input the $ICPC$ of two protocols and returns a correct converter by restricting its transitions to correct behavior for a buffer of depth B.

Algorithm 13 $Restrict(C, B)$

Input: C - the inverted complete parallel composition of two protocols.
B - buffer depth.
Output: A correct converter.

```
1: for all state q ∈ C do
2:    set Range[q] to [0..B];     // initializing to maximum range
3:    add q to Changed list;
4:    set incoming transitions ranges;
5: end for;
6: while Changed ≠ ∅ do
7:    q = pop(Changed);
8:    for all transition q' --S--> q do
9:       update transition range;
10:      Range[q'] = StateLabel(q');
11:      if Range[q'] has changed then
12:         add q' to Changed list;
13:      end if;
14:   end for;
15: end while;
```

In cases where it is impossible to construct a correct converter for the given protocols and the given buffer depth, Algorithm 13 will result in an initial state whose range does not include empty buffers - meaning that no transition is enabled at the starting point.

Notice that some transition ranges might be empty, meaning that the transition might always lead to incorrect behavior, regardless of buffer state. In such cases, if this transition can be avoided at its source state it will always be ignored, but if it cannot be ignored it will cause the state to have an empty range as well, meaning that the state should never be reached. This feature of the algorithm actually guarantees that the compatibility between the converter and the protocols is not broken, by keeping an equivalent transition in the converter to all required transitions of the protocols, covering all internal choices of the protocols.

4.2.9 Examples

Appling the *Restrict* (Algorithm 13) to the $ICPC$ of $P1$ and $P2$ yields the general converter illustrated in Figure 16. The restriction information in the transition

labels provides minimal and maximal acceptable buffer values for both data channels $(d1, d2)$ and address channels $(a1, a2)$ in the format $[d\_min, d\_max]/[a\_min, a\_max]$.

As Algorithm 13 produces the most general converter, it includes all possible correct behaviors of a converter and usually includes non-deterministic choices at different states that are possible whenever more than one transition control input are satisfied. In order to optimize the behavior of the converter by any criteria, the converter can be simplified by the removal of internal choices.

In the example of P_1 and P_2, internal choices of the converter occur in states $(0, 0), (0, 1), (0, 2)$ for most buffer states. Removal of internal choices can lead in this case to a much simpler converter, as illustrated in Figure 17.

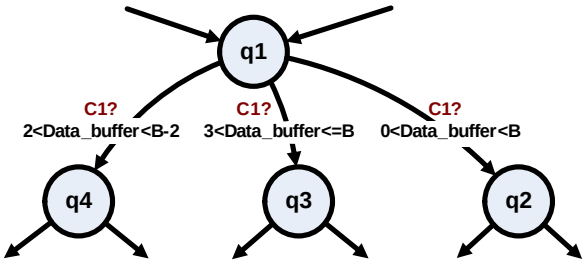

Figure 15: Complete internal choice between outgoing transitions

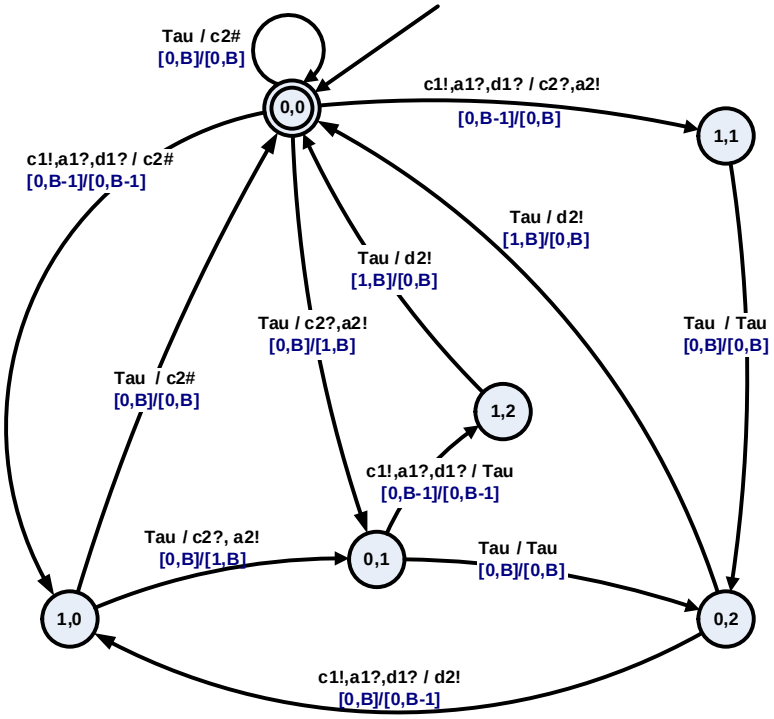

Figure 16: The most general converter for P_1 and P_2

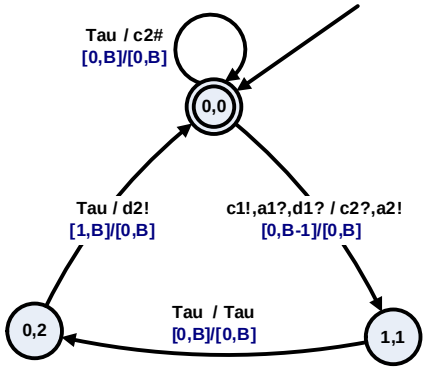

Figure 17: Minimized converter for P_1 and P_2

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 and for automatic converter synthesis which we applied to varios commercial bus protocols. 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] Open core protocol international partnership - open core protocol specification.

[2] Virtual socket interface alliance.

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

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

[5] 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.

[6] 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.

[7] ARM. Advanced micro-controller bus architecture specification.

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

[9] 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.

[10] 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.

[11] Davey, B., and Priestley, H. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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).

[17] IBM. A 32-, 64-, 128-bit core on-chip bus structure.

[18] 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.

[19] 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).

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.