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-0718
Technical Report, August 2007

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

Abstract

Hardware module reuse is a standard solution to deal with the increasing complexity of chip architectures and growing pressure to reduce time to market. In the absence of a single module interface standard, pre-designed modules for "plug and play" usually require a converter between incompatible interface protocols. Current approaches to automatic synthesis of protocol converters mostly lack formal foundations and either employ abstractions far removed from implementation or grossly simplify the structure of the protocols considered. In this work, we present a state-machine based formalism for modeling bus based communication protocols, a notion of protocol compatibility and of correct conversion between incompatible protocols. Using this formalism, we derive algorithms for checking protocol compatibility and for automatic converter synthesis. We report our experience with automatic converter synthesis between different configurations of widely used commercial bus protocols, such as AMBA AHB, ASB APB, and the open core protocol (OCP). The presented work is unique in its combination of a complete formal approach and the use of low abstraction level that enables precise modeling of protocol characteristics and simple translation to HDL.

Contents

1 Introduction **4**
1.1 Related Work 5

2 Formal Definitions **7**
2.1 Protocol Model 7
2.1.1 Modeling assumptions 9
2.1.2 Protocol Examples 9
2.2 Parallel Composition 10
2.2.1 An Example 11

3 Protocol Compatibility **14**
3.1 Defining Protocol Compatibility 14
3.2 Checking Compatibility 16
3.2.1 Examples 17

4 Converter Synthesis **21**
4.1 Defining the Converter Synthesis Problem 21

5 Automatic Converter Synthesis **23**
5.1 Complete Parallel Composition 23
5.1.1 An Example 23
5.2 Inversion of Actions 24
5.3 Restricting $ICPC$ to Correct Behavior 25
5.3.1 Restricting Transitions 26
5.3.2 Restricting States 26
5.3.3 Finding the Greatest Fixed Point: 29
5.4 Examples 30

6 Experimental Results and Conclusions **33**
6.1 Experimental Results 33
6.2 Conclusions 33

Appendices **38**

A Paths Definition: Example **38**

B Proof of Compatibility: $ICPC$ and the Protocols **40**

List of Figures

1.1 A typical SoC architecture 4

2.1 OCP master: use of counters 8
2.2 AMBA APB slave model 9
2.3 AMBA ASB bus to slave model 10
2.4 The protocols P_1 and P_2. 12
2.5 $P_1 \| P_2$, The parallel composition of P_1 and P_2 12

3.1 System structure 14
3.2 An example of allowed Read cycles 16
3.3 Protocol models 20
3.4 The parallel composition of ASB and APB 20

4.1 System structure 22

5.1 $P_1 \||| P_2$: The complete parallel composition of P_1 and P_2 25
5.2 A simple read operation 26
5.3 A simple write operation 26
5.4 No internal choice between outgoing transitions 27
5.5 Complete internal choice between outgoing transitions 27
5.6 Guard split 28
5.7 State label computation 28
5.8 Restricting transitions 29
5.9 The most general converter for P_1 and P_2 31
5.10 Minimized converter for P_1 and P_2 31
5.11 ASB to APB general converter 32
5.12 ASB to APB deterministic converter 32

A.1 The computation tree for ASB 39

List of Algorithms

1 ParallelComposition(P_1, P_2) . . . 13
2 Compatible(P_1, P_2) . . . 17
3 CheckPaths(PC) . . . 18
4 CheckPath(π, $type$) . . . 18
5 CheckMerge(π) . . . 19
6 CompleteParallelComposition(P_1, P_2) . . . 24
7 InverseTau(q) . . . 25
8 Restrict(C, B) . . . 30
9 ComputeTree(P, $initial_state$, $final_state$) . . . 39
10 IsInPath(q, T) . . . 39

Chapter 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. IP reuse enables designers to build systems using third-party modules that may comply with differing 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 including such wrappers and bridges is illustrated in Figure 1.1.

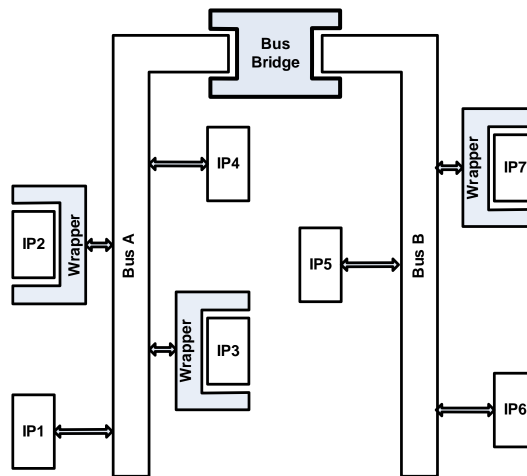

Figure 1.1: A typical SoC architecture

Though much research has been dedicated to the *converter synthesis problem* of SoC communication, converter synthesis today is still performed manually, consuming development and verification time and risking human error.

Research efforts in the field include attempts to standardize interface protocols [1, 2, 7, 17], development of 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, 26]. Attempts have been made to automate the process of generation of glue logic at different levels [3–5, 13, 14, 16, 21–23].

Different approaches and models have been investigated for automation of con-

verter synthesis: Timing diagrams [8], data queuing [15,24], message sequencing [23], and FSM based protocol modeling [3–5, 13, 14, 16, 22, 26] 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. We propose comprehensive definitions of protocol compatibility and for correct protocol converters, and derive algorithms for compatibility checking as well as converter synthesis. We report on our experiments with different commercial protocols.

1.1 Related Work

The problem of automatic converter synthesis for incompatible protocols has been addressed in the literature from different perspectives. We focus on work done in the context of hardware design using FSM-based models.

In early work [3], protocols were represented as state machines and their cross product was used to construct a converter. This work was highly innovative but preliminary, and its result was presented via a simple example. This approach was later extended in [4, 14] and is the foundation of our work.

In [13, 14] a formalism for modeling protocols using synchronous FSMs and an algorithm for wrapper synthesis are proposed. Their method extends that of [3] by distinguishing between control and data signals and explicitly considers buffering. Methods for dealing with mismatched data types and clock periods are suggested but not integrated into the given algorithm. Recent work [27] relying on [13, 14] shows that working at a higher level of abstraction reduces the size of models considered, thereby simplifying converter construction at the cost of departing from the desired hardware description converter. In [4], a product of FSMs is again used to construct protocol converters, and the product is optimized to increase bandwidth.

An alternative to converter synthesis is to use a standard communication scheme and to map disparate protocols into this scheme. For example, the scheme suggested in [25] includes an internal arbiter, and is applicable in a multi-party communication environment but imposes the restriction that each protocol be either a sender or receiver. The additional logic introduced may be significant and there is a six-cycle latency between reading and writing the *same* data item. This approach is extended in [15, 24] by using and manipulating queues for buffering data. Such *template-based solutions* are not specific to the protocols being interfaced and hence, not optimal. If the solution includes constraints such as pre-defined buffer sizes or an unavoidable latency overhead as in [25], it may not always be practical.

A third approach is to decompose protocols into smaller operations and combine operations to obtain a converter. In [20], sequential protocols are decomposed into five basic operations and a protocol behavior is organized as ordered sets of guarded executions. A converter is constructed by matching sets that transfer the same amount of data.

Passerone et al. [22] specify mismatched synchronous protocols as regular expressions and in later work [21] attempted a game theoretic formalization. The synthesis procedure is defined as a winning strategy in a game played between a

protocol and a converter and is illustrated with an example that handles reordering of data. No algorithm is presented, so it is unclear how the technique can be applied to arbitrary protocols. Recent work [26] proposes partitioning protocols into sets of transactions and synthesizing a converter by merging converters for individual transactions. Individual converters are synthesized using the method of [22] but the merging process requires manual input.

The approaches discussed above, either model protocols at a high level of abstraction, or with several simplifying restrictions, and thus, preclude completely automatic synthesis. No existing work distinguishes between control and data paths, thus the obtained converter is usually extremely large. The simple and powerful formalism proposed in this work allows, for the first time, for precise, cycle-accurate modeling of protocols and converters with distinct control and data paths. We faithfully model complicated commercial protocols and converters can easily (even automatically) be translated into HDL.

In most existing work, the definition of protocol compatibility is neglected or incorrectly assumed to be trivial, and protocol conversion is discussed without considering when such a converter is needed, or even what criteria a *correct converter* should satisfy. We introduce 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 correct protocol conversion, and propose algorithms for checking compatibility and for automatic synthesis of protocol converters.

Chapter 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. For an input control channel, a protocol can test for the presence or absence of a signal value, denoted $c?$ and $c\#$ respectively. These tests act as guards of transitions and a transition is enabled only when all of its guards are satisfied. For an output control channel, a protocol can write to (or assert) the 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 and Tau (τ) denote an empty action.

Most commercial bus protocols support *burst* operations, in which a number of continuous memory locations are either written or read, and it is a common requirement that a data item put on the bus must remain valid for more than one clock cycle. None of the existing methods for protocol modeling is expressive enough to include this notion of data repetition or its implications. We use bounded counters for this purpose. A counter is associated with each data channel and 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 is unique to the proposed formalism and 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. We denote $data!{+}{+}$, and $data!reset$, a write of a new data item in which the data counter is incremented or reset respectively, and use similar notation for a read of a new data item.

The use of counters is demonstrated in Figure 2.1, through the model of OCP master write transactions. OCP master can initiate a transaction by asserting the $MCmd$ control channel with WR value, and needs to keep both $MAddr$ and $MData$ data channels valid until $SCmdAccept$ is asserted by the slave. More detailed specification of OCP is available at [1]. The FSM in Figure 2.1 illustrates the model of OCP master while the timing diagram demonstrates a possible behavior of the model. The transitions number at the bottom of the diagram points to the transition in the model that is taken in each clock tick.

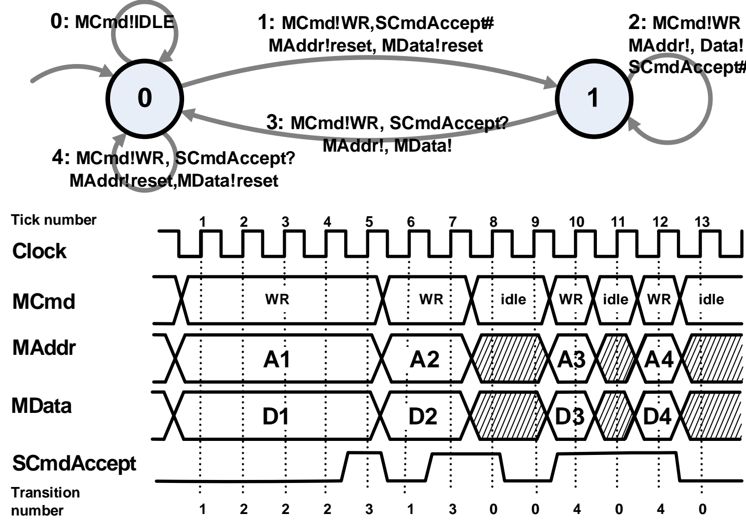

Figure 2.1: OCP master: use of counters

In order to guarantee that counter values are finite, it is a requirement of the protocols that the number of increment actions that may precede a reset is bounded. In particular, in transfers of unbounded length, the counter is reset with every data item transferred. In such cases the counter value does not represent the actual amount of data transferred but allows new data items to be distinguished from data repetition.

Definition 1 (Protocol) *A protocol P is a finite state machine with bounded counters $(Q_P, C_P, D_P, K_P, \rightarrow_P, q_s, q_f)$, where Q_P is the set of states, $C_P = C_P^I \cup C_P^O$ is a set of input and output control channels, $D_P = D_P^I \cup D_P^O$ is a set of input and output data channels, $K_P = \{k_d | d \in D_P\}$ is a set of bounded internal counters, one for each data channel ($|K_P| = |D_P|$), q_s is the initial state and q_f is the final state. Let $A_P = A_{C_P} \cup A_{D_P} \cup A_{K_P}$ be the set of actions on the control channels (A_{C_P}), data channels (A_{D_P}) and counters (A_{K_P}) of P and $\mathcal{P}(A_P)$ denote the power set of A_P. The transition relation of the protocol is $\rightarrow_P \subseteq Q_P \times \mathcal{P}(A_P) \times Q_P$. Note that all sets and relations above are finite.*

We drop the subscripts in the sets above when the context is clear. A transition (q, S, q') is denoted $q \xrightarrow{S} q'$. For a set of actions S, let $control(S)$, $data(S)$ and $counters(S)$ respectively denote the control channels, data channels and counters occurring in S.

2.1.1 Modeling assumptions

1. There is an implicit clock tick that triggers transitions. Starting at the initial state q_s, an enabled transition is taken at each clock tick.

2. Protocols do not have deadlocks or livelocks - every reachable state can reach the final state.

3. Protocols are observably deterministic over control channels—for any state $q \in Q_P$ if there exists transitions $q \xrightarrow{s1} q1$ and $q \xrightarrow{s2} q2 \in \rightarrow_P$, such that $control(s1) = control(s2)$(the transitions have the same control channel actions) $\Rightarrow s1 = s2$ and $q1 = q2$. That is, transitions from the same source state should differ in their control actions.

4. In every protocol, either $q_s = q_f$ or $q_s \xrightarrow{s} q_i \in P \Rightarrow q_f \xrightarrow{s} q_i \in P$ (The final state has transitions with the same actions and destination states as the transitions of the initial state).

2.1.2 Protocol Examples

We propose here models for an AMBA APB slave protocol and AMBA ASB bus to slave protocol.

Our model for an AMBA APB slave is illustrated in Figure 2.2. The AMBA APB protocol is defined to relate to the rising edge of the clock, and an AMBA APB slave is defined to be idle (transition from state 0 to state 0 in the model) until it is selected (by assertion of *PSEL* control channel, transitions from state 0 to state 1 and from state 0 to state 2 in the model). Once *PSEL* is asserted either a READ transaction or a WRITE transaction begins, depending on the state of control channel *PWRITE*. In a READ transaction the slave reads the address on the first cycle and writes the data on the second cycle, returning to the initial state, while in a WRITE transaction the slave can choose to read the address and data on either the first and the second cycle.

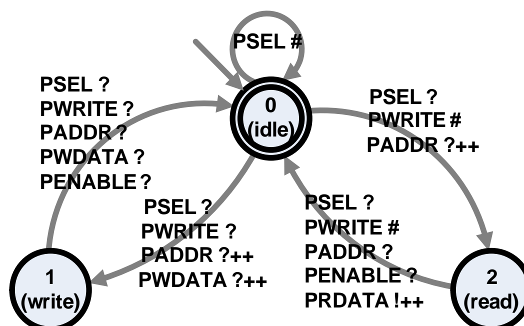

Figure 2.2: AMBA APB slave model

Our model for a write transaction of an AMBA ASB is illustrated in Figure 2.3. The AMBA ASB protocol is defined to relate to both falling and rising edges of the clock and an ASB bus can initiate a transfer with a slave by asserting the *DSEL*

control channel before a falling edge of the clock (transition from state 0 to state 2 in the model). By asserting *DSEL*, the bus initiates either a read or a write transaction, depending on the state of *BWRITE*. The duration of a transaction depends on the slave response, which can add wait states to the transaction, report an error or indicate that a memory boundary is reached (transitions outgoing state 2). Additional transactions can be pipelined as long as *DSEL* is asserted ((transition from state 4 to state 2 in the model). The model presented in Figure 2.3 details all states and transitions of a WRITE transaction, while a READ transaction model is similar, with the exceptions of *BWRITE* not being asserted and *BD* data channel being accessed with read actions rather than write actions. The transitions in dotted blue lines relate to the equivalent states in the read model and are only for low clock frequencies when there is no need for a decode cycle. More detailed descriptions of both protocols are available at [7].

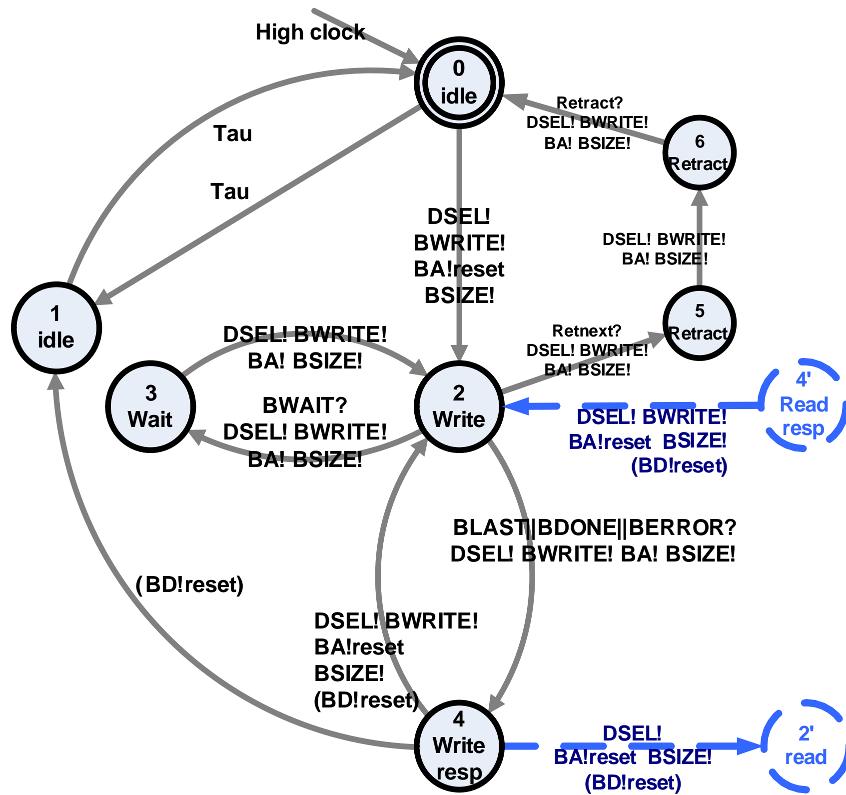

Figure 2.3: AMBA ASB bus to slave model

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 when two protocols are connected to each other and use this predicate to define the parallel composition operator.

Definition 2 (may) *The predicate* $\mathtt{may}(S_1, S_2)$ *is true for two sets of actions* S_1 *and* S_2 *iff* $\forall c \in control(S_1 \cup S_2)$*:*

$$\text{if } c? \in S_1, \text{ then } c! \in S_2, \quad \text{if } c\# \in S_1, \text{ then } c! \notin S_2,$$
$$\text{if } c? \in S_2, \text{ then } c! \in S_1, \quad \text{if } c\# \in S_2, \text{ then } c! \notin S_1.$$

Note that the definition relates to control channels only and ignores any data channel actions. The predicate $\mathtt{may}(S_1, S_2)$ is true for two sets of actions S_1 and S_2 when the two sets satisfy each other's control requirements , and therefore their transitions might be taken concurrently when the two protocols communicate. In the context of checking for protocol compatibility, we would like to check that anything that *might* happen does not violate rules of correct communication, as described in section 3.1.

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* $(q1, q2) \xrightarrow{S} (q1', q2')$ *with* $S = S_1 \cup S_2$ *is a transition of* $P_1 \| P_2$ *iff* $q1 \xrightarrow{S_1}_1 q1'$ *and* $q2 \xrightarrow{S_2}_2 q2'$ *and* $\mathtt{may}(S_1, S_2)$ *is true.*

PC describes all possible concurrent states of two protocols, subject to control compatibility, assured by `may`, and therefore includes all pairs of transitions that might be taken when the two protocols communicate with each other, reachable and unreachable from the initial state. 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. The algorithm has polynomial complexity with respect to the number of transitions in the protocols, and returns only states and transitions that are reachable from the initial state $(q1_s, q2_s)$.

2.2.1 An Example

Considering the two protocols P_1 and P_2 as presented in Figure 2.4, under a mapping of the channels $\{c1 \mapsto c2,\ a1 \mapsto a2,\ d1 \mapsto d2\}$, the parallel composition of the two protocols according to Definition 3 is illustrated in Figure 2.5. In all protocol illustrations the initial state is marked with a source-less incoming arrow, and the final state is marked with a double edge.

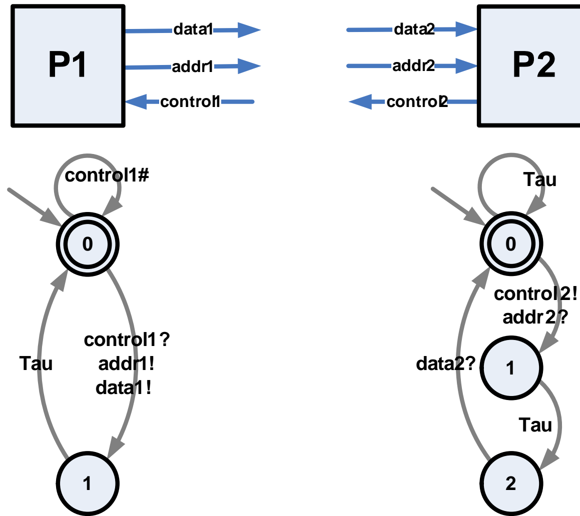

Figure 2.4: The protocols P_1 and P_2.

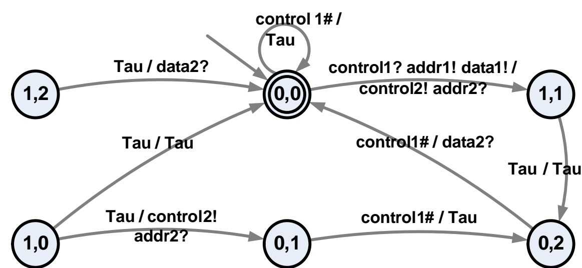

Figure 2.5: $P_1 \| P_2$, The parallel composition of P_1 and P_2

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 states and transitions 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' ∈ P1 do
4:      for all transitions q2 --S2-->_2 q2' ∈ P2 do
5:        if may(S1, S2) 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 P2
12:   end for;    // All transitions in P1
13: end for;    // All states in PC
```

Chapter 3

Protocol Compatibility

3.1 Defining Protocol Compatibility

We focus on compatibility with respect to ensuring data flow between a pair of protocols. The parallel composition of two protocols describes all possible *control 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 deadlocks can occur and livelocks can always be avoided (every reachable state is on a path to the final state).

The first requirement ensures that a data channel is accessed for a read action by a protocol only when the value of the channel is guaranteed to be valid by the other protocol (modeled as a write action), and therefore no invalid data is read. The second requirement is needed to ensure that though the same data item may be kept valid for more than one cycle, and may be read multiple times, the protocols are aware that it is the same single item. The third condition enforces that every transaction *can* terminate in a finite number of steps. Note that in order to guarantee absence of livelocks, fairness constraints need to be introduced to the protocols, which is beyond the scope of this work, and so we only require absence of deadlocks and the possibility to avoid livelocks.

For explanatory purpose, 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 3.1. We define compatibility in terms of constraints over paths in the parallel composition of two protocols.

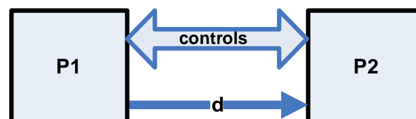

Figure 3.1: System structure

Definition 4 (*Path*) *A path π in a protocol P is a sequence of transitions $\pi = q_0 \xrightarrow{S_1} q_1 \ldots \xrightarrow{S_k} q_k$ such that for $0 \leq i < k$, $q_i \xrightarrow{S_{i+1}} q_{i+1}$ is a transition in P.*

1. *$|\pi|$ denote the number of transitions in π, also referred to as the length of π.*
2. *$New(\pi, d) = \{i \in \mathbb{N} \mid 0 < i \leq |\pi|$ and $reset(k_d) \in S_i$ or $k_d{+}{+} \in S_i\}$ be the set of indices of transitions in π in which new data is accessed on channel d.*
3. *For a path $\pi = (q1_0, q2_0) \xrightarrow{S_1} \ldots \xrightarrow{S_k} (q1_k, q2_k)$ in $P_1 \| P_2$, where $q1_j, \in P_1$ and $q2_j \in P_2$, the projection of π on P_1, $\pi \downharpoonleft P_1 = q1_0 \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_k} q1_k$. The projection $\pi \downharpoonleft P_2$ is similarly defined.*
4. *The path π is loop-free iff for all $0 \leq i < k$ and $i < j \leq k$, $q_i \neq q_j$.*
5. *The path π is a simple cycle iff $q_0 \xrightarrow{S_1} q_1 \ldots q_{k-1}$ is a loop-free path and $q_0 = q_k$.*

Let:

1. *$Paths(P, q_j, q_k)$ denote the (possibly infinite) set of paths in P from state q_j to state q_k.*
2. *$LoopFree(P, q_j, q_k)$ denote the finite set of loop-free paths in P from state q_j to state q_k.*
3. *$Cycles(P)$ denote the set of simple cycles in P.*
4. *$States(\Pi)$ be the set of states occurring in a set of paths Π.*

An example and illustration of the definition is given in Appendix A.

Definition 5 (Compatibility) *Two protocols P_1 and P_2 are compatible iff:*

1. *$Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f)) \neq \emptyset$.*
2. *For any path π from initial to final state of $P_1 \| P_2$,*
   $\pi \in Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f))$, the following hold:
   - *(a) for every transition label S, if $d? \in S$ then $d! \in S$.*
   - *(b) $|New(\pi \downharpoonleft P_1, d)| = |New(\pi \downharpoonleft P_2, d)|$.*
   - *(c) Let $i_1 < i_2 \cdots < i_n$ be the sorted sequence of indices in $New(\pi \downharpoonleft P_1, d)$ and $j_1 < j_2 \cdots < j_n$ be the sorted sequence of indices in $New(\pi \downharpoonleft P_2, d)$. For index $1 \leq \ell \leq n$ it holds that $j_{\ell-1} < i_\ell \leq j_\ell < i_{\ell+1}$ where j_0 is defined as 0 and i_{n+1} is defined as $|\pi| + 1$.*
3. *For any state $(q1, q2)$ in $P_1 \| P_2$ if $Paths(P_1 \| P_2, (q1_s, q2_s), (q1, q2)) \neq \emptyset$ then $Paths(P_1 \| P_2, (q1, q2), (q1_f, q2_f)) \neq \emptyset$.*

The first requirement for compatibility guarantees that the protocols can execute together from initial to final states. Every path from the initial to the final state should satisfy three conditions: (a) Only valid data is read. Note that unlike the `may` condition which checks only for *control channel* compatibility between transitions, here we check that every pair of transitions that *may* occur together, are also compatible in terms of *data channel* actions, and a protocol does not read data from a channel unless it is guaranteed to hold a valid value. (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, and multiple reads of an unchanged data item are not counted as distinct. An example of allowed read cycles with respect to distinct write cycles is given in Figure 3.2. Requirement 3 guarantees the absence of deadlocks and the possibility to avoid livelocks—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.

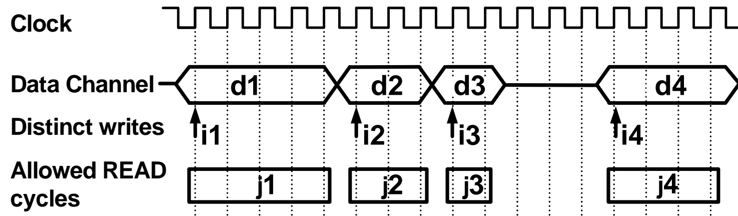

Figure 3.2: An example of allowed Read cycles

3.2 Checking Compatibility

For two protocols P_1 and P_2, compatibility is checked by Algorithm 2, by checking that the conditions of Definition 5 hold. The check that data is read only when written is made in Lines 2– 6. The absence of deadlocks and livelocks is ensured in line 7 by checking if there is a reachable state that has no path to the final state. The condition regarding sequences of reads and writes is checked using Algorithm 3.

The CheckPaths(P) algorithm is used to establish requirements 2b and 2c of Definition 5. Even though these requirements involve conditions on a potentially infinite set of paths, we reduce these conditions to conditions over two finite sets: one of loop-free paths from the initial to the final state and one of simple cycles. Lines 1–5 are used to check that every loop-free path from the initial to the final state should correctly alternate *new* read and write actions, as detailed in Algorithm 4. A similar check is made for simple cycles in Lines 6–10. However, because a cycle may begin in an arbitrary state, we also need to ensure that the sequence of reads and writes in the cycle merges correctly with those made before entering and after leaving the cycle. This check is done using Algorithm 5.

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.

1: PC = PARALLELCOMPOSITION(P_1, P_2);
2: **for all** $t \in \rightarrow$ and $d \in D$ in PC **do**
3:    **if** "$d?$" $\in t$ and "$d!$" $\notin t$ **then**
4:       **return** `False`;
5:    **end if**;
6: **end for**;
7: **if** $Paths(PC, q_s, q_f) = \emptyset \vee ($ for some $q \in Q_{PC}$:
$Paths(PC, q_s, q) \neq \emptyset \wedge Paths(PC, q, q_f) = \emptyset)$ **then**
8:    **return** `False`;   // PC has deadlocks or livelocks
9: **end if**;
10: **if** CHECKPATHS(PC) = `False`; **then**
11:    **return** `False`;
12: **end if**;
13: **return** `True`;   // P_1 and P_2 are compatible

3.2.1 Examples

For the protocols in Figure 2.4, the parallel composition demonstrated in Figure 2.5 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 2), and therefore, the two protocols will be found to be incompatible.

In order to check compatibility between ASB and APB protocols, we first need to create models that share the same clock tick. Since ASB is active on both clock edges, we extrapolate the APB model. The models for a compatibility check between an ASB initiator and APB reactor are given in Figure 3.3 (for simplicity we illustrate write transfers only).

Due to the severe incompatibility of the protocols, even with channel mapping of $PSEL \mapsto DSEL$, $PWRITE \mapsto BWRITE$, $PADDR \mapsto BA$, $PWDATA \mapsto BD$, the reachable component of the parallel composition is as illustrated in Figure 3.4. This would pass the transitions check but will fail the check that all reachable states can reach the final state, as in line 7 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 3 CheckPaths(PC)

Input: PC: a parallel composition of two protocols

```
1: for all π ∈ LoopFree(PC) do
2:   if CheckPath(π, LoopFree) = False then
3:     return False;
4:   end if;
5: end for;
6: for all π ∈ Cycles(PC) do
7:   if ( CheckPath(π, Simple) = False
          ∨ CheckMerge(π) = False ) then
8:     return False;
9:   end if;
10: end for;
11: return True;
```

Algorithm 4 CheckPath(π, $type$)

Input: a single path π of type `LoopFree` or `Simple`.

Output: `True` if the path passes the check.

```
1: π_P1 := π ⌊ P_1; π_P2 := π ⌊ P_2
2: for all d ∈ D do
   //   for every data channel in the parallel composition
   //   assuming d is input to P_2
3:   Writes := New(π_P1, d) = {i_1 < i_2 ··· < i_n1};
4:   Reads := New(π_P2, d) = {j_1 < j_2 ··· < j_n2};
5:   if d is input to P_1 then
6:     swap(Writes, Reads)
7:   end if;
8:   if (type = Simple) ∧ (j_1 < i_1) then
9:     swap(Writes, Reads)    // cycles can begin in read
10:  end if;
11:  if |Writes| ≠ |Reads| then
12:    return False;
13:  else
14:    j_0 := 0; i_{n1+1} := k + 1;
15:    for all ℓ such that 1 ≤ ℓ ≤ n1 do
16:      if (j_{ℓ−1} ≥ i_ℓ) ∨ (i_ℓ > j_ℓ) or (j_ℓ ≥ i_{ℓ+1}) then
17:        return False;
18:      end if;
19:    end for;
20:  end if;
21: end for;
22: return True;
```

Algorithm 5 CheckMerge(π)

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.

1: $Writes_\pi := New(\pi \downharpoonright P_1, d) = \{i_1 < \cdots < i_n\}$;
2: $Reads_\pi := New(\pi \downharpoonright P_2, d) = \{j_1 < \cdots < j_n\}$;
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: $Writes_{\pi 1} := New(\pi 1 \downharpoonright P_1, d) = \{i1_1 < \cdots < i1_{n1}\}$;
5: $Reads_{\pi 1} := New(\pi 1 \downharpoonright P_2, d) = \{j1_1 < \cdots < j1_{n1}\}$;
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 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 index of the last read before the merge
10: **if** $(i_1 \leq j_1) \wedge (x > y)$ **then**
11: **return** `False`; // written item never read
12: **end if**;
13: **if** $(i_1 > j_1) \wedge (x \leq y)$ **then**
14: **return** `False`; // written item read twice
15: **end if**;
16: **end if**;
17: **end for**;
18: **end for**;
19: **return** `True`;

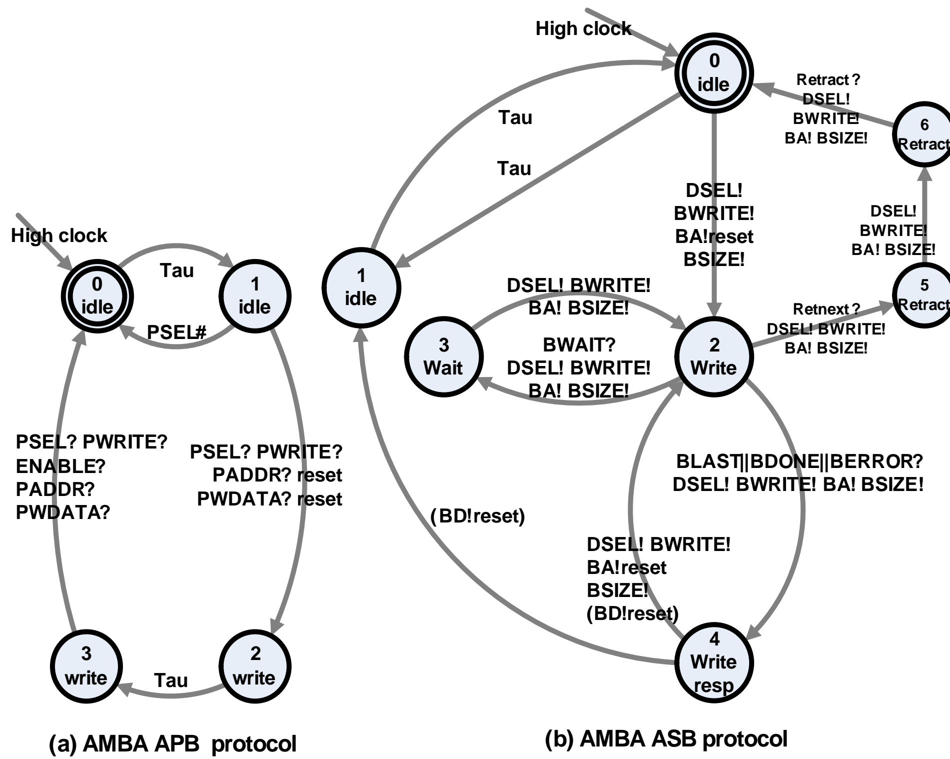

Figure 3.3: Protocol models

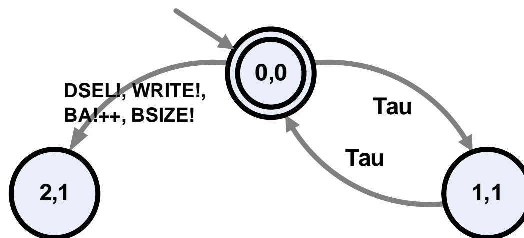

Figure 3.4: The parallel composition of ASB and APB

Chapter 4

Converter Synthesis

4.1 Defining the Converter Synthesis Problem

If two protocols are incompatible, it might still be possible to achieve correct communication between them, with the help of a protocol converter, an FSM with bounded counters and finite buffers. In a similar manner to that of direct inter module communication, to guarantee correct data flow between two protocols P_1 and P_2 communicating through a converter, the following constraints must be satisfied:

1. Data is read by a protocol (/the converter) only when written by the converter (/a protocol).
2. A data item is read as distinct 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 deadlocks can occur, and livelocks can be avoided.

In most cases, a converter needs to temporarily store data received from one protocol before writing it to the other. We assume buffers of finite depth and that information about the state of the buffers is always available to the converter. We refer to the number of data items in a buffer as a *data state* of the converter, as opposed to the *control state*, that describes the pairs of states of the protocols.

For explanatory purpose, 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 4.1. We also assume a finite buffer of depth B for storage of data items.

Definition 6 (Converter Synthesis Problem) *Given two incompatible protocols P_1 and P_2, synthesize a protocol M satisfying the following:*

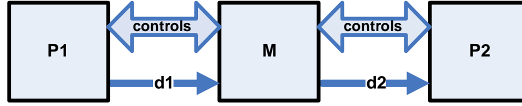

Figure 4.1: System structure

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 any $1 \le \ell \le n$ it holds that $i_\ell \le j_\ell \le i_{\ell+B}$.

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 final states and all buffers are empty.

The first two requirements guarantee that data items are read as distinct exactly once, only when written and that no deadlocks can be reached and livelocks can be avoided (corresponding to the first, second and fifth constraints for correct data flow). The third requirement relates to a notion of fidelity on behalf of the converter—guaranteeing that the converter passes all given data and does not create 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), as either would result in data corruption. In the general case where there are more data channels, constraint 3 of Definition 6 should hold for every pair of mapped channels independently.

Chapter 5

Automatic Converter Synthesis

In this section, we present an algorithm for the construction of a correct converter. This Algorithm takes as input, two protocols and a map between data channels, and returns the most general correct converter—describing all possible correct behaviors of a converter, out of which smaller and deterministic converters can be extracted. The algorithm includes the following stages: (1) construction of a complete parallel composition as described in section 5.1, (2) inversion of actions (section 5.2), and (3) restriction to correct behavior (section 5.3).

5.1 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 7 (Complete Parallel Composition (CPC)) *The complete 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 and $S = S_1 \cup S_2$.*

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

An algorithm for the construction of a complete parallel composition is given in Algorithm 6. The algorithm has polynomial complexity with respect to the number of transitions in the protocols, and computes the entire CPC that by definition does not contain any states that are not reachable from the initial state $(q1_s, q2_s)$.

5.1.1 An Example

Considering the two protocols P_1 and P_2 as presented in Figure 2.4, the complete parallel composition of the two protocols according to Definition 7 is illustrated in Figure 5.1.

Algorithm 6 COMPLETEPARALLELCOMPOSITION(P_1, P_2)

Input: Two Protocols
Output: $CPC = P_1 ||| P_2$ 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

5.2 Inversion of Actions

All outputs of the protocols need to be used as inputs to the converter and vice-versa. Thus, we need to invert all actions of the CPC in order to construct a converter. This is done by replacing every action with its complementary action.

Complementary actions are defined in Table 5.1, 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).

Table 5.1: 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	$c\#, \forall c \in C^O$

A minimized complementary action for Tau, for transitions outgoing of a state q in a protocol, can be calculated using Algorithm 7, as only control output channels that are referred by other outgoing transitions in the protocol need to be addressed.

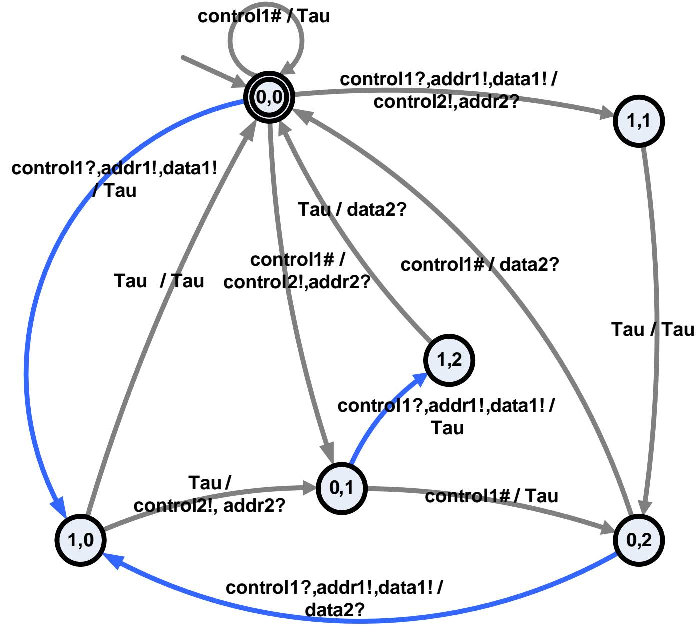

Figure 5.1: $P_1 \parallel\!| P_2$: The complete parallel composition of P_1 and P_2

5.3 Restricting $ICPC$ to Correct Behavior

The inverted CPC ($ICPC$) satisfies requirements 1 and 2 of Definition 6, because each path in the converter is an inverted projection of a path in each of the protocols. A formal proof of compatibility is available in Appendix B. It remains to constrain the $ICPC$ to comply with requirement 3 to arrive at a correct converter. The $ICPC$ is restricted by avoiding transitions that may lead to a violation of condition 3, Definition 6. Since this condition is concerned with avoiding buffer overflows and underflows, each transition is augmented with a condition on the converter's data state, to avoid an overflow or underflow of buffers.

Algorithm 7 InverseTau(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;

5.3.1 Restricting Transitions

Let *Data_State* be the number of items in the buffer. A read action when the buffer is already full ($Data_State = B$) will cause a buffer overflow. To avoid a possible overflow we add a condition to transitions with read actions that prevent taking the transition if the buffer is full as illustrated in Figure 5.2.

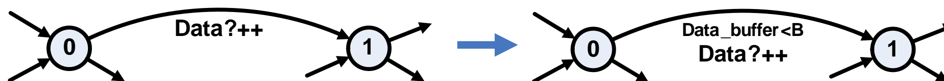

Figure 5.2: A simple read operation

Similarly, a write action made when the buffer is empty ($Data_State = 0$) will result in a buffer underflow. Such transitions are avoided by adding a condition to transitions that avoids taking such a write action if the buffer is empty, as shown in Figure 5.3.

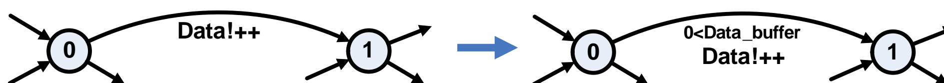

Figure 5.3: A simple write operation

These restrictions are added under the assumption that any data state is possible in the destination control state (labeled 1 in Figure 5.3). That is, every control state can be reached at a data state of range $0 \leq Data_State \leq B$. In general, the data states associated with a control state may have a smaller range, say: $0 \leq x \leq Data_State \leq y \leq B$. A control state should not be reached in a data state that is outside this range, so incoming transitions to a control state must be restricted.

For a write action, in a transition whose destination control state has a data state range of $x \leq Data_State \leq y$ the transition restriction is:

$$min(x + 1, B + 1) < Data_State < min(y + 1, B + 1),$$

and for a read action, the restriction is:

$$max(-1, x - 1) < Data_State < max(-1, y - 1).$$

Such conditions are added to all transitions. 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.

5.3.2 Restricting States

A condition imposed on a transition may affect the deadlock possibilities in the source state for that transition. If the possible data states in a given control states do not satisfy the constraints of any outgoing transitions from that state, a deadlock will occur. The occurrence of such a situation depends on the *internal non-determinism*,

or *internal choice* of a state. That is, the extent to which the transition from a state is defined by input control.

For example, the conditions on transitions in Figure 5.4 are mutually exclusive, so the data state in control state $q1$ must be consistent with the conditions of *all* outgoing transitions to avoid a deadlock. That is the admissible data states in the control state $q1$ are the conjunction of the conditions on the outgoing transitions: $3 < Data_State < B - 2$.

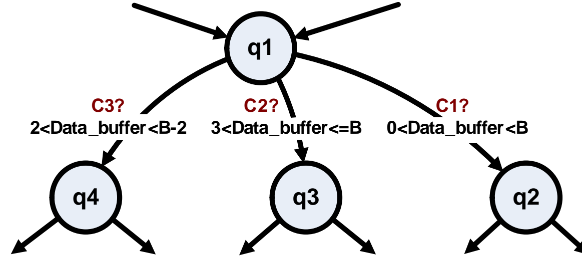

Figure 5.4: No internal choice between outgoing transitions

In contrast, the conditions on transitions from the control state $q1$ in Figure 5.5 are not mutually exclusive, hence there is some internal choice available to the protocol about which transition to choose. The admissible data states in such a control state are the disjunction of conditions on outgoing transitions: $0 < Data_State \leq B$.

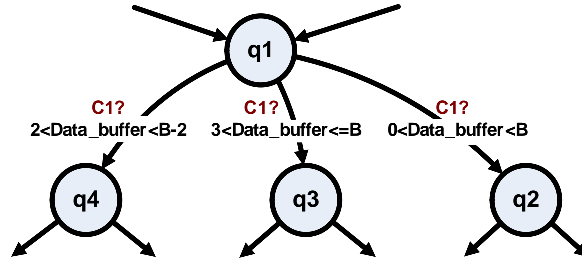

Figure 5.5: Complete internal choice between outgoing transitions

The range of admissible data states in a given control state is computed by examining its outgoing transitions. Each transition has a guard—a set of input control actions, and a data state condition. There may be internal choice between transitions from a state due to: 1) replication: a guard appearing in multiple transitions. or 2) under-specified guards: actions on some input control channels are absent in a guard. An under-specified guard corresponds to several mutually exclusive guards as illustrated in Figure 5.6.

The range of data states in a control state is determined in two steps. (1) Each under-specified guard is replaced by the corresponding mutually exclusive guards. (2) Computing first, the disjunction of the conditions on replicated guards and then the conjunction of conditions between sets of mutually exclusive guards. For the state in Figure 5.7 the outgoing transitions are replaced by the following:

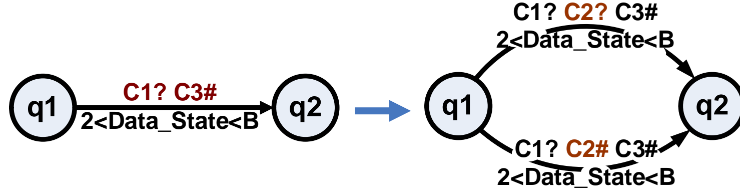

Figure 5.6: Guard split

$$q1 \xrightarrow[0<Data_State<B]{c1?,c2\#,c3?} q2, \quad q1 \xrightarrow[0<Data_State<B]{c1?,c2\#,c3\#} q2,$$

$$q1 \xrightarrow[3<Data_State\leq B]{c1?,c2?,c3\#} q3, \quad q1 \xrightarrow[3<Data_State\leq B]{c1\#,c2?,c3\#} q3,$$

$$q1 \xrightarrow[2<Data_State<B-2]{c1?,c2?,c3?} q4, \quad q1 \xrightarrow[2<Data_State<B-2]{c1?,c2\#,c3?} q4$$

The range of data states for $q1$ is computed as follows:

$$\begin{aligned}
&((0 < Data_State < B) \vee (2 < Data_State < B-2)) \\
&\wedge (0 < Data_State < B) \wedge (3 < Data_State \leq B) \\
&\wedge (3 < Data_State \leq B) \wedge (2 < Data_State < B-2) \\
&= 3 < Data_State < B-2
\end{aligned}$$

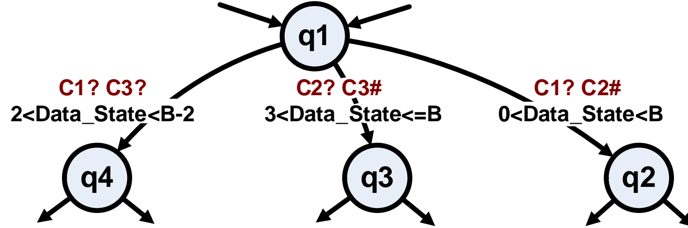

Figure 5.7: State label computation

To obtain a correct converter, the data state conditions on transitions and control states must be repeatedly updated until a consistent set of conditions is obtained. An illustration of this process over a sequence of three write actions is provided in Figure 5.8, where the intervals $[x, y]$ in red show the guards and data constraints updated in each application. In sub-figure (a) the initial restrictions are added to transitions, in (b) the control state ranges are updated and in (c) a second iteration begins with adjustments of the transition restrictions to the ranges of destination states. Sub-figure (d) illustrates the fixed point of this example. Expectedly, such a sequence should not be reached unless there is room to store at least three data items.

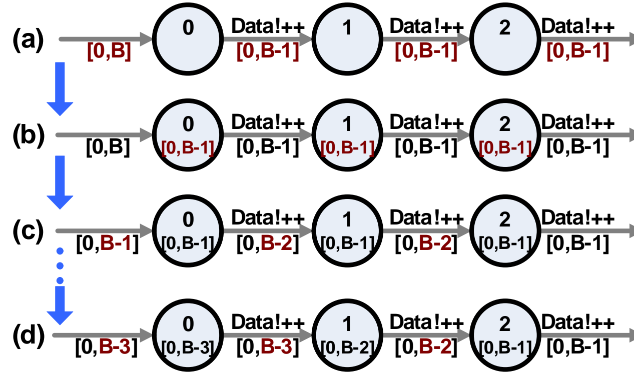

Figure 5.8: Restricting transitions

5.3.3 Finding the Greatest Fixed Point:

The function of updating the range of data states in a control state, is monotonic over the closed set $[0..B]$. When applied iteratively, the range obtained in an iteration is bounded by the previous range. If initialized to the range [0,B], these characteristics guarantee that the update procedure converges after a finite number of iterations [11] to the greatest fixed point of the update function—the largest range of data states allowed in each control state.

Algorithm 8 takes as input the $ICPC$ of two protocols and returns a correct converter that uses a buffer of size B. Lines 1–5 in Algorithm 8 initialize the control state labels to the maximal data state range and set the initial transition conditions. Lines 6–15, iteratively update these labels and conditions until a fixed point is reached.

In cases where it is impossible to construct a correct converter with the specified buffer size for the given protocols, Algorithm 8 will result in either an initial state in which the buffer is not empty—meaning that no transition is enabled at the starting point, or a final state with non-empty buffers—meaning that some data transfer may never end. The Algorithm complexity is bounded by $O(B \times N)$, where B is the size of buffers and N is the number of transitions in the $ICPC$.

Notice that the range of data states on some transitions may be empty, meaning that the transition always leads to incorrect behavior, regardless of data state. In such cases, if this transition can always be avoided from its source state it can be removed. If it cannot be avoided (say due to deadlock possibilities) the range of data states in the source state will be empty as well, meaning that such a control state should never be reached. This feature of the algorithm guarantees the absence of deadlocks in the converter (by keeping an equivalent transition in the converter to all required transitions of the protocols, covering all internal choices of the protocols), but does not guarantee absence of livelocks. However, livelocks might only be created due to internal nondeterminism, and therefore, if found, a livelock can always be removed from the converter without breaking the compatibility of the converter with the protocols. A check for absence of livelocks needs to be performed separately.

Algorithm 8 RESTRICT(C, B)

Input: C - the *ICPC* of two protocols. B - buffer depth.
Output: A correct converter.

```
1:  for all state q ∈ C do
2:     set Range[q] to [0..B];
3:     add q to Changed list;
4:     set incoming transitions ranges;      // as in subsection 5.3.1
5:  end for;
6:  while Changed ≠ NIL do
7:     q = head(Changed);
8:     for all transition q' -S-> q do
9:        update transition range;      // as in subsection 5.3.1
10:       Range[q'] = compute_state_label(q');      // as in subsection 5.3.2
11:       if Range[q'] has changed then
12:          add q' to Changed list;
13:       end if;
14:    end for;
15: end while;
```

5.4 Examples

Applying RESTRICT(Algorithm 8) to the *ICPC* of $P1$ and $P2$ in Figure 2.4 yields the general converter in Figure 5.9. The restriction information in the transition labels provides minimal and maximal acceptable data states for both data channels $(d1, d2)$ and address channels $(a1, a2)$ in the format $[d_min, d_max]/[a_min, a_max]$.

The most general converter produced by Algorithm 8 is correct but may include internal choice. It can be simplified by removing certain transitions contributing to internal choice. 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 5.10.

In the example of ASB and APB protocols, the general converter is illustrated in Figure 5.11, with the exception of transitions related to retraction (states 5 and 6 in ASB) since there is no use to these states. Figure 5.12 demonstrates a smaller deterministic correct converter.

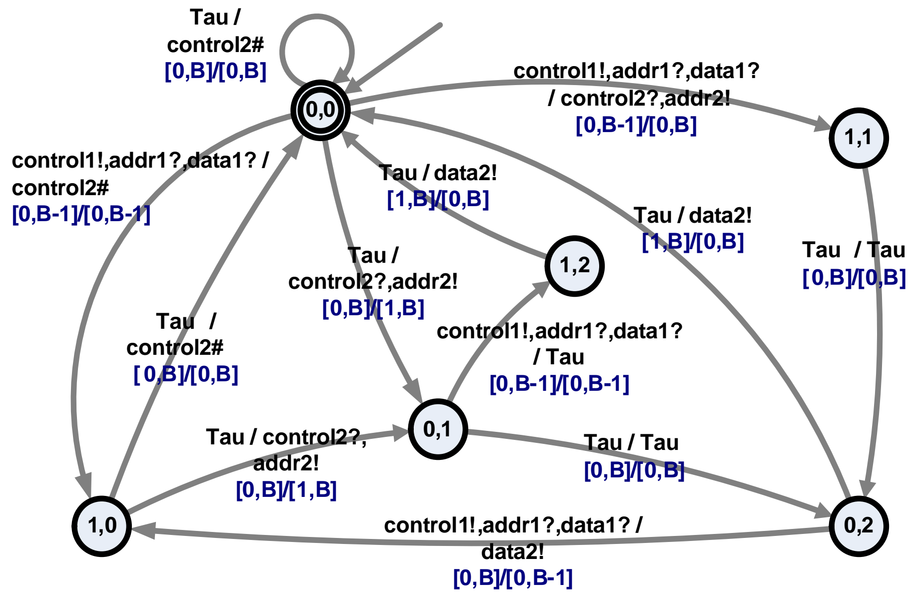

Figure 5.9: The most general converter for P_1 and P_2

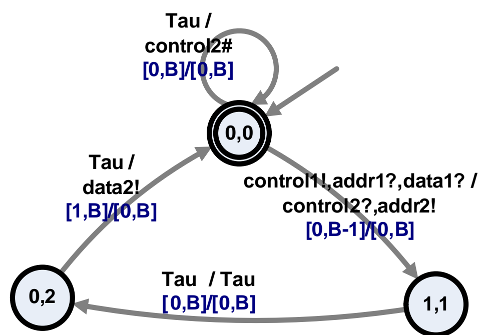

Figure 5.10: Minimized converter for P_1 and P_2

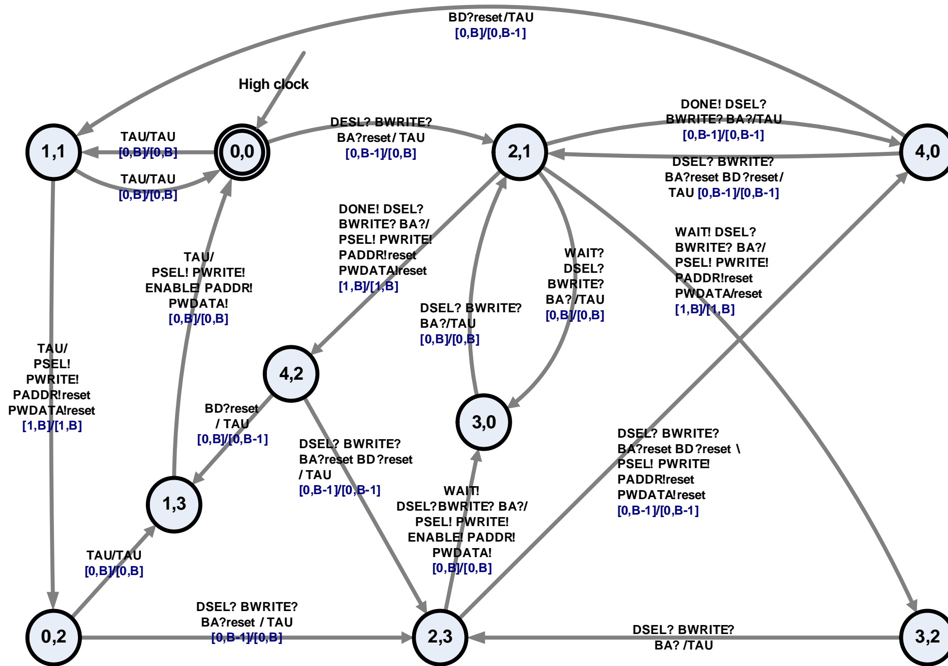

Figure 5.11: ASB to APB general converter

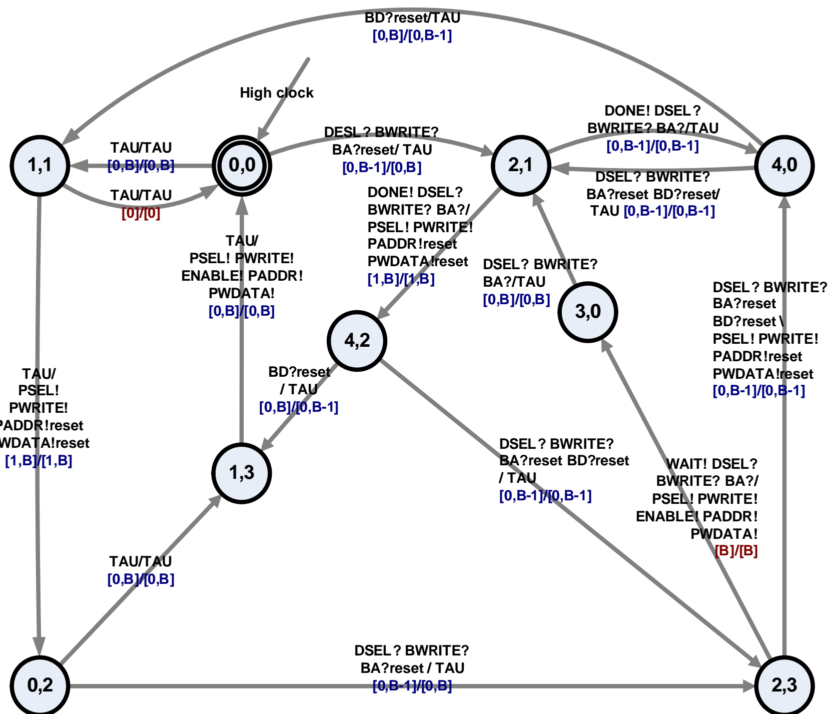

Figure 5.12: ASB to APB deterministic converter

Chapter 6

Experimental Results and Conclusions

6.1 Experimental Results

In our experiments, we chose to focus on the Advanced Micro controller Bus Architecture (AMBA) family of protocols that are very widely used in the industry and are diverse in performance level and complexity and cover many protocol features [7]. Using the presented formalism we have successfully modeled the protocols of the Advanced Peripheral Bus (APB), Advanced System Bus (ASB) and Advanced High-performance Bus (APB), as well as some configurations of the Open Core Protocol (OCP) [1], modeling master, slave and bus specifications in unprecedented detail, for both read and write operations.

The presented algorithms have been implemented in a PC based tool and applied to pairs of protocol models, at different bandwidth ratios. The experiments conducted for automatic converter synthesis for incompatible protocols are listed in Table 6.1, where the numbers next to the protocol name represents number of states in the protocols' model. In all listed experiments a converter was successfully created, with the exception of abort operations that are currently not covered but will be added in future work. It is a property of our converter synthesis algorithm that the number of states in a synthesized converter is bounded by the product of the number of states of its protocol models, an improvement by orders of magnitude to existing methods.

6.2 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 various commercial bus protocols. We have demonstrated the process

Table 6.1: Automatic converter synthesis - experimental results

Protocols		Conditions tested				
		BWidth ratio			Operations	
Initiator	Reactor	1:1	1:2	2:1	Read	Write
ASB(12)	APB(7)	✓	✓	✓	✓	✓
APB(7)	ASB(10)	✓	✓	✓	✓	✓
AHB(8)	APB(3)	✓	✓	✓	✓	✓
APB(3)	AHB(6)	✓	✓	✓	✓	✓
APB(3)	OCP(5)	✓	✓	✓	✓	✓
OCP(5)	APB(3)	✓	✓	✓	✓	✓
ASB(12)	OCP(9)	✓	✓	✓	✓	✓
OCP(9)	ASB(10)	✓	✓	✓	✓	✓

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.

Bibliography

[1] Open core protocol international partnership - open core protocol specification, http://www.ocpip.org.

[2] Virtual socket interface alliance, http://www.vsi.org/.

[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, http://www.arm.com/.

[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, http://www-03.ibm.com/chips/products/coreconnect.

[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), 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] Watanabe, S., Seto, K., Ishikawa, Y., Komatsu, S., and Fujita, M. Protocol transducer synthesis using divide and conquer approach. In *ASP-DAC* (2007), IEEE.

[27] Yun, C., Bae, Y., Cho, H., and Jhang, K. Automatic synthesis of interface circuits from simplified ip interface protocols. In *ACSAC* (2006), Springer, pp. 581–587.

Appendix A

Paths Definition: Example

For a general protocol P that has cycles, $Paths(P, q_s, q_f)$ is an infinite set, made from a finite set of loop-free paths from initial to final state (denoted $\texttt{LoopFree}(P, q_j, q_k)$), and a finite set of simple cycles (denoted $\texttt{Cycles}(P)$).

For example, in the model of a AMBA ASB as illustrated in Figure 3.3(a), Every path from the initial state (state label 0) to the final state (state label 0) is a combination of the set $\texttt{LoopFree}(ASB, 0, 0)$ and $\texttt{Cycles}(ASB)$, where:

$$\begin{aligned}
\texttt{LoopFree}(ASB, 0, 0) &= \{ \quad \pi_1 = 0 \xrightarrow{Tau} 1 \xrightarrow{Tau} 0, \\
&\qquad \pi_2 = 0 \xrightarrow{S_1} 2 \xrightarrow{S_2} 4 \xrightarrow{S_3} 1 \xrightarrow{Tau} 0, \\
&\qquad \pi_3 = 0 \xrightarrow{S_1} 2 \xrightarrow{S_4} 5 \xrightarrow{S_5} 6 \xrightarrow{S_6} 0 \; \} \\
\texttt{Cycles}(ASB) &= \{ \quad \pi_4 = 2 \xrightarrow{S_7} 3 \xrightarrow{S_8} 2, \\
&\qquad \pi_5 = 2 \xrightarrow{S_2} 4 \xrightarrow{S_9} 2 \},
\end{aligned}$$

S_1 = “DSEL! BWRITE! BA!reset BSIZE!”,

S_2 = “BLAST‖BDONE‖BERROR? SDEL! BWRITE! BA! BSIZE!”,

S_3 = “BD!reset”,

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!reset BSIZE! BD!reset”,

In order to extract the sets of loop free paths and possible simple 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 ($\leq |Q_P|$).

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. The computation tree of ASB is presented in Figure A.1, and an algorithm for the construction of the tree is given in Algorithm 9.

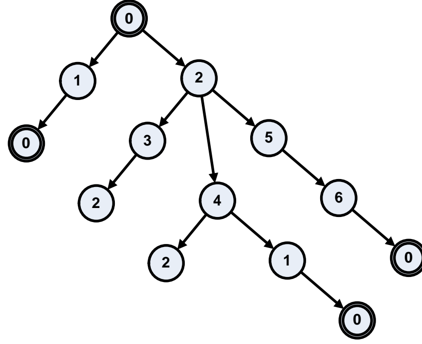

Figure A.1: The computation tree for ASB

Algorithm 9 COMPUTETREE(P, $initial_state$, $final_state$)

1: $Level[0] = initial_state$;; $i = 0$; // Levels counter
2: **while** $Level[i] \neq \emptyset$ **do**
3: **for all** states $q \in Level[i]$ **do**
4: **if** $(i == 0)$ or ((q is not in path) and ($q \neq final_state$)) **then**
// (see Algorithm 10 for finding whether q is in the path)
5: **for all** outgoing transitions $q \xrightarrow{S} q'$ **do**
6: add q' to $Level[i+1]$;
7: add $q \xrightarrow{S} q'$ to the tree;
8: **end for**;
9: **end if**;
10: **end for**;
11: $i++$;
12: **end while**;

Algorithm 10 ISINPATH(q, T)

1: **if** $q == T.root$ **then**
2: return TRUE
3: **else**
4: $to_check = q$;
5: **while** $to_check \neq T.root$ **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**;

Appendix B

Proof of Compatibility: $ICPC$ and the Protocols

Compatibility between $ICPC$ and a protocol P_i is decided by checks over the parallel composition $ICPC \| P_i$. Without loss of generality, it is sufficient to show that $ICPC$ is compatible with only one of the protocols.

To show that $ICPC$ is compatible with protocol P_1, We first show that every state in the reachable part (with respect to the initial state $(q1_s, (q1_s, q2_s)))$ of the partial parallel composition of P_1 and $ICPC$ ($P_1 \| ICPC$, Definition 3) is of the form $(q1_i, (q1_i, q2_j))$ where $q1_i \in P_1$ and $q2_j \in P_2$:

Since the initial state, $(q1_s, (q1_s, q2_s))$, is of the required form, for a state not of the required form to be reachable, there has to be a transition from a "well formed" state $(q1_a, (q1_a, q2_b))$ to a "not well formed" state : $(q1_a, (q1_a, q2_b)) \stackrel{s1}{\rightarrow} (q1_i, (q1_k, q2_j))$ $(q_i \neq q_k \in P_1))$.

The existence of transition $(q1_a, (q1_a, q2_b)) \stackrel{s1}{\rightarrow} (q1_i, (q1_k, q2_j))$ in $ICPC \| P_1$ implies the following:

a)	$(q1_a, q2_b) \stackrel{s2}{\rightarrow} (q1_k, q2_j) \in ICPC$	by definition of $ICPC \| P_1$.
b)	$q1_a \stackrel{s1_1}{\rightarrow} q1_i \in P_1$	by definition of $ICPC \| P_1$.
c)	$\texttt{may}(s1_1, s2) = \texttt{True}$	by definition of $ICPC \| P_1$.

Furthermore, A implies the following:

a.1)	$q1_a \stackrel{s2_1}{\rightarrow} q1_k \in P_1$	by definition of $ICPC(P_1, P_2)$.
a.2)	$q2_b \stackrel{s2_2}{\rightarrow} q2_j \in P_2$	by definition of $ICPC(P_1, P_2)$.
a.3)	$s2 = \texttt{inverse}(s2_1) \cup \texttt{inverse}(s2_2)$	

We now show that the existence of the transitions (a.1) and (b) contradicts assumption 3 of determinism in protocols structure (subsection 2.1.1):

$s2 = \texttt{inverse}(s2_1) \cup \texttt{inverse}(s2_2)$:
By definition of $\texttt{inverse}$, $\forall c \in C^I_{P_1}$:
$c? \in s2_1 \Rightarrow c! \in s2$
$c\# \in s2_1 \Rightarrow c! \notin s2$

By definition of $\texttt{inverse}$, $\forall c \in C^O_{P_1} \subseteq C^I_{ICPC}$:
$c! \in s2_1 \Rightarrow c? \in s2$
$c! \notin s2_1 \Rightarrow c\# \in s2$

and since either $c! \in s2_1$ or $c! \notin s2_1$, it implies that for all control channels, that are in $C^O_{P_1}$ either $c? \in s2$ or $c\# \in s2$ (in $ICPC$).

On the other hand, $\mathtt{may}(s1_1, s2) = \mathtt{True}$:
By definition of $\mathtt{may}(s1_1, s2)$ $\forall c \in C^I_{P_1}$:
$c? \in s1_1 \Rightarrow c! \in s2$
$c\# \in s1_1 \Rightarrow c! \notin s2$
By definition of $\mathtt{may}(s1_1, s2)$ $\forall c \in C^O_{P_1} \subseteq C^I_{ICPC}$:
$c? \in s2 \Rightarrow c! \in s1_1$
$c\# \in s2 \Rightarrow c! \notin s1_1$

This implies that the control actions of $s1_1$ and $s2_1$ are the same, which contradicts assumption 3.

$\Rightarrow$ every *state* in the reachable part of $P_1 \| ICPC(P_1, P_2)$ is of the form: $(q1_i, (q1_i, q2_j))$ where $q1_i \in P_1$ and $q2_j \in P_2$.

$\Rightarrow$ every *transition* in the reachable part of $P_1 \| ICPC(P_1, P_2)$ is of the form: $(q1_i, (q1_i, q2_j)) \xrightarrow{s} (q1_m, (q1_m, q2_n))$, and for $q1_i \xrightarrow{s1} q1_m \in P_1$ and $q2_j \xrightarrow{s2} q2_n \in P_2$, $s = s1 \cup \mathtt{inverse}\{s1\} \cup \mathtt{inverse}\{s2\}$.

To show that $ICPC$ is compatible with P_1 we follow the definition of compatibility, referring to the partial parallel composition $P_1 || ICPC$:

Condition 1: Included in condition 3. See there.

Condition 2a: Since all transitions in $P_1 || ICPC$ have labels of the form $s = s1 \cup \mathtt{inverse}\{s1\} \cup \mathtt{inverse}\{s2\}$, then by definition of $s1 \cup \mathtt{inverse}\{s1\}$, for all $d \in D_{P_1}$, $d? \in s1 \Leftrightarrow d! \in \mathtt{inverse}\{s1\}$ and $d2 \in \mathtt{inverse}\{s1\} \Leftrightarrow d! \in s1$

Conditions 2b and 2c: By definition of $\mathtt{inverse}\{s1\}$, $\mathtt{inverse}\{k{+}{+}\} = k{+}{+}$ $\Rightarrow i_1 < i_2 \cdots < i_n = j_1 < j_2 \cdots < j_n$.

Condition 3: Since all states of $P_1 || ICPC$ are of the form $(q1_i, (q1_i, q2_j))$ (as proved above), proving condition 3 for $P_1 || ICPC$ is the same as proving it for $ICPC$. Condition 3 requires that every state that is reachable from the initial state should also be able to reach a final state. For a state $(q1_i, q2_j)$ in $ICPC$ to be reachable from $(q1_s, q2_s)$ there need to be a path in P_1 from $q1_s$ to $q1_i$ and a path in P_2 from $q2_s$ to $q2_j$ such that the two paths have the same length (k).

$$\pi_1 = q1_s \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_k} q1_i \in P_1$$

$$\pi_2 = q2_s \xrightarrow{S2_1} q2_1 \ldots \xrightarrow{S2_k} q2_j \in P_2$$

Since all states in the protocols can reach the final state of the protocol (assumption 2), there exist paths of lengths m and n:

$$\pi_3 = q1_i \xrightarrow{S1_{k+1}} q1_{i+1} \ldots \xrightarrow{S1_{k+m}} q1_f \in P_1$$

$$\pi_4 = q2_j \xrightarrow{S2_{k+1}} q2_{j+1} \ldots \xrightarrow{S2_{k+n}} q2_f \in P_2$$

Concatenating π_1 with π_3 and similarly π_2 with π_4 yields a path of length $k+m = M$ in P_1, that reaches state $q1_i$ after its k transition, and a path of length $k+n = N$ in P_2that reaches state $q2_j$ after its k transition.

$$\pi_{1,3} = q1_s \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_k} q1_i \xrightarrow{S1_{k+1}} q1_{i+1} \ldots \xrightarrow{S1_{k+m}} q1_f \in P_1$$

$$\pi_{2,4} = q2_s \xrightarrow{S2_1} q2_1 \ldots \xrightarrow{S2_k} q2_j \xrightarrow{S2_{k+1}} q2_{j+1} \ldots \xrightarrow{S2_{k+n}} q2_f \in P_2$$

Using assumption 4, there exists paths (cycles) of lengths M and N:

$$\pi_{1f} = q1_f \xrightarrow{S1_1} q1_1 \ldots \xrightarrow{S1_k} q1_i \xrightarrow{S1_{k+1}} q1_{i+1} \ldots \xrightarrow{S1_{k+m}} q1_f \in P_1$$

$$\pi_{2f} = q2_f \xrightarrow{S2_1} q2_1 \ldots \xrightarrow{S2_k} q2_j \xrightarrow{S2_{k+1}} q2_{j+1} \ldots \xrightarrow{S2_{k+n}} q2_f \in P_2$$

Concatenating $\pi_{1,3}$ with $(N-1)$ concatenations of the cycle π_{1f} yields a path of length $M + M \times (N-1) = M \times N$ in P_1 whose k state is $q1_i$ and the $M \times N$ state is $q1_f$.

Concatenating $\pi_{2,4}$ with $(M-1)$ concatenations of the cycle π_{2f} yields a path of length $N + N \times (M-1) = M \times N$ in P_2 whose k state is $q2_j$ and the $M \times N$ state is $q2_f$.

$\Rightarrow ICPC$ has a path beginning at the initial state $(q1_s, q2_s)$ of length $M \times N$ whose k state is $(q1_i, q2_j)$ and the $M \times N$ state is $(q1_f, q2_f)$.

$\Rightarrow ICPC$ has a path beginning at the initial state $(q1_i, q2_j)$ of length $(M \times N - k)$ whose $(M \times N - k)$ state is $(q1_f, q2_f)$.