A Formal Approach to Interface Synthesis for System-on-Chip Design

Vijay D'silva Arcot Sowmya Sridevan Parameswaran
Email : {vijayd,sowmya,sridevan}@cse.unsw.edu.au
School of Computer Science and Engineering
The University of New South Wales

S. Ramesh
Email : ramesh@cse.iitb.ac.in
Center for Formal Design and Verfication of Software
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

UNSW-CSE-TR-0304 — April 2003

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

Abstract

Systems-on-Chip (SoC) design methodologies rely increasingly on reuse of intellectual property (IP) blocks. IP reuse is a labour intensive and time consuming process as IP blocks often have different communication interfaces. We present a framework to generate a synthesizable VHDL description of an interface between two mismatching IP communication protocols. We improve and extend previously published work by formalising the problem and by explicitly handling data width and type mismatching and multiple data transfers. At present, simpler cases of pipelining are handled as well. We have implemented our technique and demonstrate it by generating an interface between the CoreConnect Processor Local Bus from IBM and the AMBA System Bus from ARM.

1 Introduction

1.1 Motivation

The current VLSI design scenario is characterised by high performance, complex functionality and short time-to-market. A reuse-based methodology for SoC design has become essential in order to meet these challenges. Typically, a SoC is an interconnection of different pre-verified IP blocks which communicate using complex protocols. Integration of such blocks would usually require some glue logic to be inserted between them. Approaches adopted to facilitate IP integration include the development of a few standard on-chip bus architectures such as the CoreConnect from IBM and the AMBA from ARM and the work of the VSI Alliance[9]. Unfortunately, the vision of assembling SoCs using plug-and-play IP blocks is yet to become a reality for various reasons[3] including:

- Lack of a single standard bus architecture resulting in IPs still being designed to interact with different protocols.
- Integration of IP blocks into an SoC is largely a manual process requiring considerable effort.
- Verification of the entire system is still a bottle neck due to interface and timing issues.

Interface synthesis is an area of research[5] that seeks to automate the process of interconnecting components at different levels. Low level interfaces are primarily concerned with physical quantities such as voltage and capacitance while at higher levels they address abstract behaviours such as communication between processes or state machines. We focus on the synthesis of interfaces which facilitate the exchange of data.

1.2 Related Work

The interface synthesis problem has been addressed in the literature and the following causes of protocol mismatch have been identified: differences in clock speeds, signalling conventions, sequencing of data and data width or data type mismatches. Additional issues arising in interface construction are optimising latency and buffer sizes, resolving non-determinism and preserving timing requirements.

Borriello and Katz[4] use timing diagram specifications of protocols to construct event graphs and then generate a logic circuit which behaves as a transducer. Their technique requires that data buses have the same names and that the designer provide the information for correct merging of event graphs.

Gajski et al.[14] decompose a protocol specification into a combinations five basic operations, and organise the protocol behaviour as ordered sets of guarded executions. Sets transferring the same amount of data are matched an interface is constructed. Data width, control and clock signal mismatches are handled but the sequencing of data must be the same in both protocols.

Smith and De Micheli[19] map any given protocol into a standard communication scheme which identifies between *senders* and *receivers*. Their scheme can be applied in a multi-party communication environment involving components operating at different frequencies. Their technique is not optimized for timing as data has to be transferred between buffers, amount of control logic as an internal arbiter is used, and storage as separate input and output buffers are used. Further, it is assumed that data types are matching.

Passerone et al.[17] use regular expression based specifications of synchronous protocols to synthesize an interface which uses only a single buffer and minimizes the latency between transfers. The synthesis algorithm cannot be easily extended to different kinds of data type mismatching, and does not handle multiple transfers. As stated in[16] the methods reviewed so far

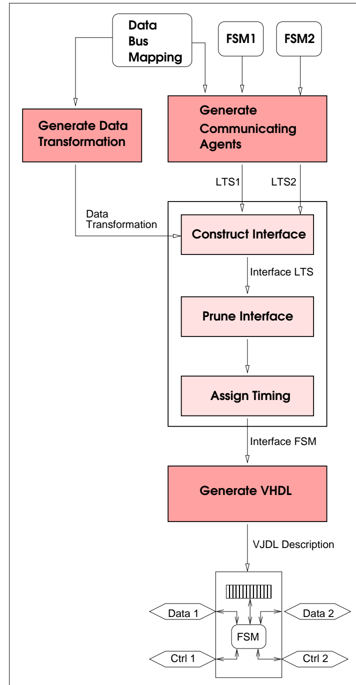

Figure 1: Design Flow for Interface Synthesis

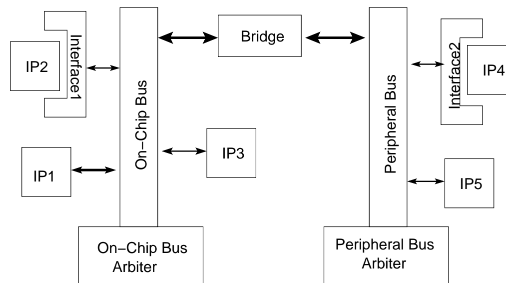

Figure 2: On-Chip Bus Architecture with Interfaces and mismatching IPs

lack a mathematically sound formalisation and interpretation. Consequently, it is unclear how the techniques developed apply to any two arbitrary mismatching protocols.

Recently, de Alfaro et al.[16] formalised the problem in a game theoretic framework. They identify the functions of an interface as a type converter handling data value mismatches and a protocol converter resolving other interaction mismatches. The interesting question of whether an interface exists is equated with the existence of a winning strategy for the game. The interface synthesized is synchronous and is capable of modifying the timing and order of data but data value translation is not incorporated in the synthesis methodology as it will result in state explosion.

Interface synthesis has been addressed in the context of network protocols as the protocol conversion problem. Okumura[15] uses finite automata to model network protocols a product based approach to construct an interface given a conversion seed. Calvert and Lam[6] perform the synthesis using a service specification with progress and safety properties to ensure correctness. Tao et al.[20] take protocol and channel specifications as input and derive constraints which are used to compute an optimal interface in the context of layered network protocols. While the problem is similar and the treatment rigorous, significant contextual differences prevent easy application of these techniques to mismatching hardware protocols.

1.3 Paper Contributions and Overview

In this paper, we propose a formal framework for the protocol mismatching problem and provide a solution capable of handling mismatching data widths, types and multiple transfers. In particular, the formalisation is inspired by Milner's well known theory of communicating systems[12] which provides an intuitive interpretation for communicating Finite State Machines(FSMs). We focus on modelling data and observe that the synthesis procedure varies depending on the data transformation to be performed. A set of interface construction rules are proposed which may be modified to incorporate various data transformations.

The modelling and synthesis flow is shown in Figure 1. The designer is expected to provide protocol specifications as Mealy style FSMs(henceforth referred to as FSMs). Manufacturers usually provide protocol and bus architecture specifications as FSMs or timing diagrams which are easily translatable into FSMs. We use FSMs as an input because they capture all the required behaviour and simultaneously minimize the designer's effort. FSMs are handled within our framework and used to synthesize an interface using a data transformer and a set of construction rules. Then, dead states are pruned and timing information is assigned to generate a VHDL description of the interface. We also provide a new method for checking protocol compatibility and are able to prove that the interface behaves as required. Figure 2 illustrates the use of interfaces with on-chip buses in a typical reuse scenario when protocol mismatching is encountered. Our ideas have been implemented and applied to two on-chip bus protocols commonly used in SoC design. Our preliminary results are very encouraging.

The paper is organised as follows. Section 2 introduces the preliminary definitions, the notion of compatibility and the problem specification. Section 3 contains a formal description of an interface and synthesis rules. Section 4 demonstrates the interface synthesized for a mismatch between the IBM CoreConnect Processor Local Bus and ARM's AMBA Peripheral Bus and we conclude in Section 5 with a discussion and indicate directions for future work.

2 Protocol Specification and Modelling

2.1 Definitions

When provided with protocol specifications, IP blocks may be designed with the functional and communication components interleaved or separate as advocated in [1, 18]. The latter approach is more conducive to reuse. These protocols may be expressed as Mealy style FSMs[8] which are formalised below.

Definition 1 *A Mealy machine is a state machine* $M = (Q_M, I_M, O_M, \sigma_M, \lambda_M, q_{M0})$. Q_M *is the state space,* I_M *and* O_M *are the input and output alphabet.* $\sigma_M \subseteq Q_M \times I_M \times Q_M$ *and* $\lambda_M \subseteq Q_M \times I_M \times O_M$ *are the state transition and output functions respectively. A transition is written as* $q \xrightarrow{a/b}_M q'$ *where* $a \in I$ *and* $b \in O$.

Most systems perform many read or write actions in a single transition hence a and b in a transition would be replaced by $S_a \subseteq I_M$ and $S_b \subseteq O_M$ respectively. Given a FSM specification, we define a communicating agent which is more weildy than an FSM for interface synthesis.

Definition 2 *A* communicating agent *is defined as a tuple* $P = (Q_P, \Sigma_P, \Delta_P, V_P, \mathcal{A}_P, \longrightarrow_P, q_0)$ *where*

- $Q = Q_{Pt} \cup Q_{Pu}$ *is a finite set of timed and untimed states.*
- Σ_P *is a set of control channels.*
- Δ_P *is a set of data channels.*

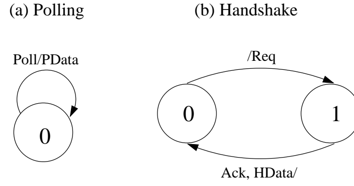

Figure 3: Mealy machines of the Polling and Handshake protocols

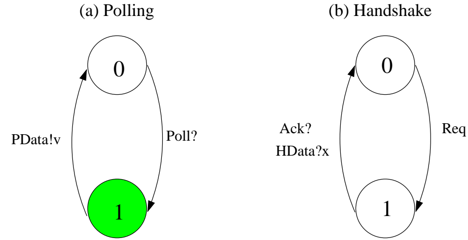

Figure 4: Communicating Agents for Polling and Handshake Protocols

- *V_P is a set of variables such that there exists one variable corresponding to each data channel.*
- *$\mathcal{A}_P = Pow(\Lambda^r) \cup Pow(\Lambda^w) \cup \{\tau\}$ is a set of operations. Each operation is a set of atomic read(Λ^r), write(Λ^w) or internal (τ) actions. $Pow(A)$ denotes power set of A.*
- *$\longrightarrow_P \subseteq Q_P \times \mathcal{A}_P \times Q_P$ is the transition relation.*
- *q_0 is the initial state.*

The subscripts will be dropped when the context is clear. Intuitively, Q_{Pt} contains the states the protocol might be in at the beginning of each clock cycle; control channels model signals, data channels model buses and atomic actions are reads or writes performed on channels. $c!$ causes an event on a control channel and $c?$ denotes a read on a control channel and can be performed only if an event has occurred on that channel. Control actions model the raising and detection of the value of a signal. Signals which are active low can be modelled by a channel c_z. Control signals consisting of more than one wire are modelled similar to one-hot encoding used in HDL coding. If c is a data channel, $c!v$ denotes the value v being written on c and $c?x$ denotes that the value on c is read into a buffer x.

An operation is a set of actions and captures that in a given clock cycle many actions can be performed. $\mathcal{A}^r = Pow(\Lambda^r)$ denotes the set of read operations and write operations are denoted $\mathcal{A}^w = Pow(\Lambda^w)$. Operations belonging to these two sets are *external* while the action τ is *internal*. τ is used to model the passage of time in the absence of external activity. We write $\mathcal{A}_C$ to refer to operations performed only on channels in the set C. Every external action a has a complementary action denoted $\bar{a}$. A read and write on the same channel are complementary actions. The complement of an operation S_a is $\bar{S_a} = \{\bar{a} | a \in S_a\}$. A transition is written as $q \xrightarrow{S_a}_P q'$ and denotes that the operation S_a is performed in the transition from q to q'. $DataChannels(S_a)$ refers to the data channels operated on in S_a.

Figure 3 contains Mealy machine descriptions of a polling and a handshake protocol. Figure 4 shows the corresponding communicating agents with the untimed states shaded. A communicating agent P is obtained from a Mealy machine M as follows: for every transition $r \xrightarrow{S_a/S_b}_M r'$ there exists a pair of transitions $q \xrightarrow{S'_a}_P q'' \xrightarrow{S'_b}_P q'$ in P such that q'' is an untimed state. For every $c \in S_a$ there exists a $c? \in S'_a$ if c is a control channel and $c?x$ if c is a data channel. Similarly,

the output set is replaced by a write operation. As a consequence of constructing communicating agents from FSMs, we observe that untimed states have only one incoming and outgoing transition and have no transitions to other untimed states. Hence sequences of transitions of type Zeno[11] (infinite activity occuring in finite time) are disallowed.

2.2 Protocol Compatibility

Given two protocols, we say they are compatible if they are capable of executing successfully together. We formalise this idea by adapting the definition of bisimulation[12] to communicating agents.

Definition 3 *A pair of communicating agents P_1 and P_2 are* matching *if there exists a binary relation $\mathcal{R} \subseteq Q_{P_1} \times Q_{P_2}$ over the states of P_1 and P_2 such that*

1. *$r_0 \mathcal{R} s_0$ where $r_0 \in P_1$ and $s_0 \in P_2$ are initial states of P_1 and P_2*
2. *For any pair of states r and s such that $r\mathcal{R}s$ the following hold:*
   (a) *If $r \xrightarrow{S_a} r'$ and $S_a \neq \{\tau\}$ then there exists some s' such that $s \xrightarrow{\bar{S}_a} s'$ and $r'\mathcal{R}s'$.*
   (b) *If $r \xrightarrow{\tau} r'$ and $r, r' \in Q_{P_1 t}$, there exists some s' such that $s \xrightarrow{\tau} s'$ satisfying that $s, s' \in Q_{P_2 t}$ and $r'\mathcal{R}s'$.*
3. *$\mathcal{R}$ is a symmetric relation.*

The definition requires that at every stage in its execution, if one agent can perform an external operation or a finite wait, the other agent should be able to perform the complementary operation or wait equally long. A protocol pair (P_1, P_2) that does not satisfy these requirements is said to be *mismatching*.

2.3 Operation of Communicating Agents and Interfaces

We now describe the operation of the machines described so far. The operation of a single communicating agent is identical to that of the Mealy Machine it is generated from. The operation of a pair of communicating agents may be described by the parallel composition operator $||$. A pair of matching agents P_1 and P_2 in states r and s respectively,denoted $\langle r, s\rangle$, when composed as $(P_1||P_2)$ will make the transitions $r \xrightarrow{S_a} r'$and $s \xrightarrow{S'_a} s'$ synchronously if $S'_a = \bar{S}_a$. The transition is labelled with the set S_c of channels on which actions are performed and denoted $\langle r, s\rangle \xrightarrow{S_c} \langle r', s'\rangle$. An agent in a protocol pair in state r can perform a read operation only if the complementary operation is performed by the other agent. If the complementary write is not performed and there is a transition $r \xrightarrow{\tau} r$ the agent is capable of waiting for the required operation to be performed. If such a transition does not exist and no other transition is possible, the agent will deadlock. It may be observed that the composition $(P_1||P_2)$ will perform all operations successfully only if P_1 and P_2 are matching.

The composition of an interface with a communicating agent is similarly defined. The set of operations of an interface I defined over a pair of communicating agents P_1 and P_2 may be written as $\mathcal{A} = A_{P_1} \cup A_{P_2} \cup \{\tau\}$. The composition $(I||P_2)$ will make a transition $\langle q, s\rangle \xrightarrow{S_a} \langle q', s\rangle$ if $S_a \in \mathcal{A}_{P_1}$ and $\langle q, s\rangle \xrightarrow{S_c} \langle q', s'\rangle$ if $S_a \in \mathcal{A}_{P_2}$, $q \xrightarrow{S_a} q'$, $s \xrightarrow{S'_a} s'$ and $S'_a = \bar{S}_a$. Informally, transitions which involve operations on channels shared with P_2 require the complementary operation to be performed in order to take place while those operations on channels shared with P_1 are included with no requirement.

The framework developed so far allows for a definition of the problem. *Given a mismatching protocol pair (P_1, P_2) synthesize an interface I such that $(P_1, (I||P_2)$ form a matching protocol pair.*

3 Interface definition and Synthesis

The interface between two mismatching protocols mentioned above is a device which should have the following properties

1. Passivity: the interface does not initiate any transactions.
2. Two-Phased: It should be able to read from and write to the channels of both P_1 and P_2 and distinguish between them.
3. Consistency: Data which has been read must be output to the required channel after performing the required data transformation. Further, the interface should not be a source or a sink of data.

3.1 Interface Definition

A generic schematic of an interface is shown in Figure 1 and is formalised below.

Definition 4 *An* interface *between two mismatching protocols is a machine* $I = (Q, \Sigma_1, \Delta_1, \Sigma_2, \Delta_2, V, \mathcal{A}, \longrightarrow, q_0)$ *such that*

- $Q = Q_t \cup Q_u$ *is a set of timed and untimed states.*
- $\Sigma_1, \Delta_1, \Sigma_2, \Delta_2$ *are distinct sets of control and data channels.*
- V *is a set of variables.*
- $\mathcal{A}$ *is a set of operations.*
- $\longrightarrow$ *is the state transition relation.*
- $q_0 \in Q_t$ *is the initial state.*

Information regarding the correspondence between data channels of two protocols has to be provided by the designer and cannot be determined automatically. Two data channels might either have mismatching data width or data type or both. The former is resolved by fragmentation or combination of data and the latter by devising a type transformer. When data is read, it may be transformed into the output format and stored in a buffer till it is required. Alternately it may be stored in a buffer and the type transformation performed prior to output. The choice between the two is made depending on the optimum combination of storage and time required to perform a type transformation.

Formally, a correspondence between data channels is a bijective function $f : \Delta_1 \rightarrow \Delta_2$. Let $D(c)$ be the symbolic data type of a data channel. $D(c)$ might simply describe the width of a data channel or provide more complex information. The definition of a data transformer follows.

Definition 5 *A* data transformer $\mathcal{T} : (c, f(c)) \rightarrow W$ *is a mapping defined for every* $c \in \Delta_1$ *such that* $f(c) \in \Delta_2$ *and* $W \subseteq D(c) \times D(f(c))$.

Given a pair of corresponding channels $(c_i, f(c_i))$, the transformation indicates how data from the domain of one is mapped to the domain of the other. In the case of mismatching bus widths, computing the number of packets from one bus which correspond to the other is simple arithmetic. This information is denoted as a set $N^{\mathcal{T}} = \{(n_i, n'_i)\}$ of the number of the reads which must be performed on c_i to successfully transfer data over $f(c_i)$ via a sequence of n'_i writes. We will use a set of pairs of counters $N = \{(y_i, y'_i)\}$ intialised to these values in the

construction process to ensure the data consistency requirements are not violated. N^0 denotes that all the counters are zero. $N^m_{y_i}$ denotes that the value of counter y_i is set to m. Let the function $g(c_i)$ return the counter variable y_i for c_i if $c_i \in \Delta_1$ and y'_i if $c_i \in \Delta_2$.

It is reasonable to assume that the two bus widths will be multiples of each other as widths are usually designed to be powers of 2. The rules which will follow can handle mapping between bus widths of arbitrary sizes, but more attention will have to be paid to buffer operations.

3.2 Interface Construction: Transitions

The interface is defined in terms of the protocol pair (P_1, P_2) whose communication it is to facilitate. Elements of P_1 and P_2 are subscripted while those of I are not. N denotes the set of counters and $g(c)$ is as defined previously. $r \in Q_{P_1}$, $s \in Q_{P_2}$, .

- $Q \subseteq \{[r, s, N]\}$ is the set of states.
- $\Sigma_1 = \Sigma_{P_1}, \Sigma_2 = \Sigma_{P_2}, \Delta_1 = \Delta_{P_1}, \Delta_2 = \Delta_{P_2}$.
- V is such that there exists one variable x_i for each pair of channels $(c_i, f(c_i))$ and $size(x_i) = max(size(c_i), size(f(c_i)))$.
- $[r_0, s_0, N^{\mathcal{T}}]$ is the initial state.
- Given two transitions $r \xrightarrow{S_a}_{P_1} r'$ and $s \xrightarrow{S'_a}_{P_1} s'$ we define $\longrightarrow$ for I as follows. k and l are indexes of positions in the buffer to which data must be written. These values have to be calculated prior to each use in a rule and are computed easily so details will not be given here.

  1. If $S_a = S'_a = \{\tau\}$ then $[r, s, N] \xrightarrow{\tau} [r', s', N]$.
  2. If $S_a \in A_{P_1 \Sigma_1}$ then $[r, s, N] \xrightarrow{\bar{S_a}} [r', s, N]$.
  3. If $S_a \in \mathcal{A}^w$ and for all $c_i \in DataChannels(S_a)$, if $\prod_i g(c_i) \neq 0$ then for all such c_i, $\bar{a_i} = c_i?x_i[k : l]$, $N' = N^{g(c_i)-1}_{g(c_i)}$. Add the transition $[r, s, N] \xrightarrow{\bar{S_a}} [r', s, N']$
  4. If $S_a \in \mathcal{A}^r$ and for all $c_i \in DataChannels(S_a)$, if $\prod_i g(c_i) \neq 0$ and $\sum_i g(f(c_i)) = 0$ then for all such c_i, $\bar{a_i} = c_i!x_i[k : l]$, $N' = N^{g(c_i)-1}_{g(c_i)}$. Add the transition $[r, s, N] \xrightarrow{\bar{S_a}}$ $[r', s, N']$
  5. Rules 2,3 and 4 are symmetrically defined for transitions in P_2.

Rule 1 ensures that τ actions are performed only if both agents can perform them thereby maintaining synchrony. Rule 2 inserts a complementary operation for every control operation. The counters are not modified as no data operation is performed. Rule 3 handles interface read operations on one or more data channels. If the required counters are not zero ($\prod_i g(c_i) \neq 0$) the reads are performed and the relevant counters are decremented. In Rule 4 when the interface has to output data, an additional check is performed ($\sum_i g(f(c_i)) = 0$) to ensure that valid data can be output.

It is important to note that in the rules given, only decrement operations are performed on counters. Hence, the final state reached will have all counters at 0 and they will have to be explicity reinitialised. Similarly, if the counters relating to the channels of one agent are zero and it has reached its initial state, no further transitions from that agent should be used in construction. To meet these and the passivity requirement of the interface, we impose the follow conditions on the rules above.

1. No write transition are made from the initial state.

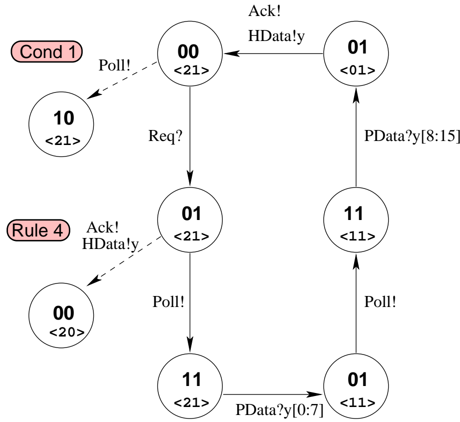

Figure 5: Interface between Polling and Handshake Protocols

2. All transitions to the state $[r_0, s_0, N^0]$ are redirected to the state $[r_0, s_0, N^{\mathcal{T}}]$.

3. If the interface is in state $[r_0, s, N]$ and $\forall c_i \in \Delta_1, g(c_i) = 0$ no transitions of P_1 are considered in transitions from this state. Expectedly, this condition is symmetric.

Figure 5 shows the interface which has been constructed between the handshake and polling protocols. The correspondence is $f(\texttt{PData}) \rightarrow \texttt{HData}$; `PData` is an 8 bit bus and `HData` a 16 bit one. The transformation $\mathcal{T} : (c, f(c)) \rightarrow (2, 1)$ so $N^{\mathcal{T}}$ and N contain only one pair. The labels of the states in Figure 5 indicate the corresponding states of the two protocols and the value of the counter is shown in angular brackets. The dotted transitions are not included as they violate the requirements of the rule or condition indicated. The transition back to the initial state is as per the requirement of condition 2.

Looping behaviour is handled by introducing a set of predicates and related counter operations $\{(p_i, Incr_i(N))\}$ such that there is one predicate for each transition causing a loop. If a predicate evaluates to true in a given transition, the related increment operation is performed.

The rules ensure that the interface synthesized incorporates only valid behaviours but may still include dead states. Dead states are those which do not have a path to the initial state and arise when one of the protocols contains transitions to an error state. They can be eliminated by performing a backward reachability from the initial state and marking all states encountered. All unmarked states and the transitions leading to them can then be removed.

3.3 Interface Construction : Timing

The steps given so far generate the transitions of the interface. It remains to identify timed states to enable a FSM and consequently HDL code to be generated. The assignment of timed states should be such that in one clock cycle the interface performs at most the operations of the two protocols it interfaces. Product construction might introduce nondeterminism or pseudo-nondeterminism[17] into the interface when a single read operation precedes different write operations. Nondeterminism can sometimes be resolved by choosing one of the different paths available. In a situation where protocols support single and back to back transfers, a given initial input may produce different behaviours which are determined by some internal operation resolution maynot be possible. A choice between different paths can be made by depending on whether storage or latency is to be optimized.

In order to decide timing, each state of the interface is labelled with a tuple (i, j), indicating the number of clock cycles consumed by each protocol. A label might be a number designating a clock cycle or the symbol u followed by a number indicating that a clock value is undefined in the current state, and capturing the last defined clock value.

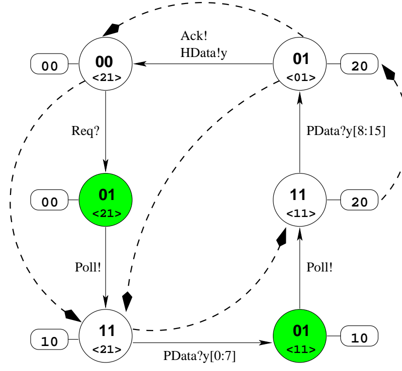

Figure 6: Marking of timed and untimed states

1. $[r_0, s_0, N^{\mathcal{T}}].label = (0, 0)$

2. For a state $[r, s, N]$ with label (i, j) such that there exists a transition $[r, s, N] \xrightarrow{S_a} [r', s', N']$, If

   (a) S_a is a read operation in $\mathcal{A}_{\Sigma_1} \cup \mathcal{A}_{\Delta_1}$ then

      i. if i is a number, $[r', s', N'].label = (u; i, j)$.

      ii. if $i = u; i'$ where i' is a number then,
      $[r, s, N].label = (i' + 1, j)$ and
      $[r', s', N'].label = (u; i' + 1, j)$.

   (b) S_a is a write operation in $\mathcal{A}_{\Sigma_1} \cup \mathcal{A}_{\Delta_1}$ then

      i. if i is a number, $[r', s', N'].label = (i + 1, j)$.

      ii. if $i = u; i'$ where i' is a number then,
      $[r, s, N].label = (i', j)$ and
      $[r', s', N'].label = (i' + 1, j)$.

The rules are symmetric and follow from the observation that a read operation always begins in a timed state and that a write operation always terminates in a timed state. The symbol u is required to deal with sequences of transitions which interleave read and write operations.

We now distinguish between timed and untimed states. The initial state is a timed state. Timing for the remaining states is assigned by considering every sequence of transitions $[r, s, N] \xrightarrow{S_{a_1}} \dots \xrightarrow{S_{a_n}} [r', s', N']$ such that $[r, s, N] \in Q_t$. If $[r, s, N].label = (i, j)$ and $[r', s', N'].label = (i', j')$ then $[r', s', N'] \in Q_t$ if the following conditions are satisfied :

1. $i' \leq i + 1$ and $j' \leq j + 1$

2. There exists no pair $(m, n) \in N$ such that the corresponding pair $(m', n') \in N'$ satisfies $(m', n') = (m - 1, n - 1)$

The first condition ensures that the interface progresses only one clock cycle with each pair of clock cycles in the two protocols. The second condition states that a data packet cannot be read and written by the interface within the same clock cycle. In Figure 6, the states of the interface are labelled with tuples attached to each state and the untimed states have been shaded. The dotted transitions comprise final FSM as shown in Figure 7. The FSM can be translated into VHDL code and relevant analysis may be performed using CAD tools.

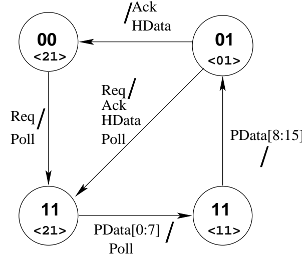

Figure 7: Mealy Machine of the interface

3.4 Correctness of Generated Interface

The interface which is generated is correct if when composed with P_2 it forms a matching protocol with P_1 and vice versa.

Theorem 1 *If P_1 and P_2 are mismatching protocols, the interface I which is synthesized ensures that $(P_1, (I||P_2))$ is a matching protocol pair.*

Proof Sketch 1 *The machine $(I||P_2)$ maybe written as an operationally identical product machine. This machine is a projection of the actions of I on channels shared with P_1. We show that the only external actions of $(I||P_2)$ are those on channels of P_1. Then, it has to be shown that all τ actions are matched as required completing the two requirements of matching.*

Proof 1 *The machine $(I||P_2) = (Q, \Sigma, \Delta, \mathcal{A}, \longrightarrow, \langle[r_0, s_0, N], s_0\rangle)$ where*

- $Q = \{\langle[r, s, N], s\rangle\}$
- $\Sigma = \Sigma_1$*, the control channels of the interface which are not shared with* P_2.
- $\Delta = \Delta_1$*, the data channels not shared with* P_2
- $\mathcal{A} = A_1 \cup \{\tau\}$ *where* $\mathcal{A}_1$ *is the set of operations on channels* Σ_1 *and* Δ_1.
- $\longrightarrow$ *follows from the definition of* $||$.

We define a relation $\mathcal{R}$ such that every pair of states $r \in Q_{P_1}$ and $\langle[r, s, N], s\rangle \in Q, r\mathcal{R}\langle[r, s, N], s\rangle$. It remains to show that $\mathcal{R}$ satisfies the requirements of a matching relation.

1. $r_0\mathcal{R}\langle[r_0, s_0, N], s_0\rangle$ *as required.*
2. *For every pair of states such that* $r\mathcal{R}\langle[r, s, N], s\rangle$
   (a) *For every transition* $r \xrightarrow{S_a} r'$*, we have that* $\langle[r, s, N], s\rangle \xrightarrow{\bar{S_a}} \langle[r', s, N'], s\rangle$ *due to the construction of I and definition of* $||$.
   (b) *For every transition,* $r \xrightarrow{\tau} r'$ *there exists* $\langle[r, s, N], s\rangle \xrightarrow{\bar{\tau}} \langle[r', s, N], s\rangle$ *due to the interface construction process.*
3. *The relation satisfied by $2(a)$ is symmetric because for every action in P_1, there exists a complementary action in I. Symmetry holds as these are the only external operations in $(I||P_2)$.* □

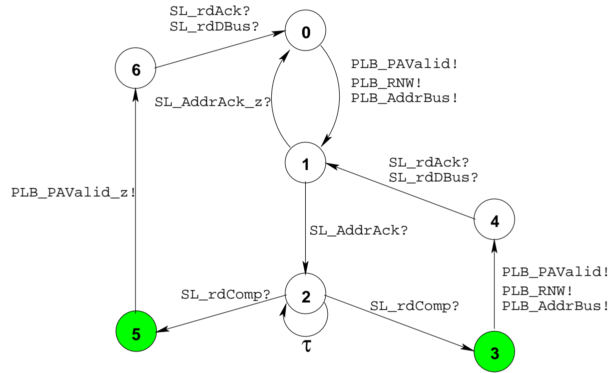

Figure 8: IBM CoreConnect Processor Local Bus Specification

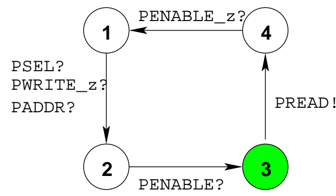

Figure 9: AMBA Peripheral Bus Specification

Gate Count	855
Flip Flop Count	49
Max. Operating Frequency	357MHz
Peak Memory Usage	92MB

Table 1: Physical Characteristics of the Synthesized Interface.

```
process (clk)
begin
if (clk'event and clk = '1') then
 case state is
  when 0 =>
    if (PLB_PAValid = '1' and PLB_RNW = '1') then
      data1buffer <= PLB_AddrBus;
      state <= 1;
    end if;
  when 1 =>
      SL_AddrAck <= '1';
      state <= 2;
  when 2 =>
      SL_rdComp <= '1';
      PSEL <= '1';
      PWRITE_z <= '1';
      PADDR <= data1buffer;
      state <= 3;
```

```
when 3 =>
 if (PLB_PAValid = '1') then
   data1buffer <= PADDR;
   PENABLE <= '1';
   state <= 4;
 end if;
 if (PLB_PAValid_z = '1') then
   PENABLE <= '1';
   state <= 5;
 end if;
when 4 =>
   data2buffer <= PREAD;
   state <= 6;
when 5 =>
   data2buffer <= PREAD;
   state <= 7;
```

```
when 6 =>
  SL_rdDBus <= data2buffer;
  SL_AddrAck <= '1';
  PENABLE_z <= '1';
  state <= 1;
when 7 =>
  SL_rdDBus <= data2buffer;
  SL_AddrAck <= '1';
  PENABLE_z <= '1';
  state <= 0;
when others =>
    state <= 0;
  end case;
 end if;
end process;
```

Figure 10: VHDL Code

4 Experimental Results

A tool which takes FSMs as input and generates VHDL interfaces has been implemented. Input can be provided as text in the `dot` format or using the graphical editor `dotty` both of which are part of a set of graph visualisation tools[7].

We have applied the methodology presented to model the communication interface of IBM's CoreConnect Processor Local Bus(PLB)[10] and ARM's AMBA Peripheral Bus(APB)[2] and synthesize an interface between them. FSM descriptions of the protocols were made from the specification documents. Figures 8 is a communicating agent modelling the single and back to back read transfer behaviour of PLB and 9 models the read behaviour of the APB. It can be seen that the operation of the APB is much simpler than that of the PLB. Infact, the PLB is a highspeed bus while the APB being a peripheral bus operates at lower frequencies.

The data channels correspond as follows: ($\texttt{PLB_AddrBus} \rightarrow \texttt{PADDR}$) and ($\texttt{PREAD} \rightarrow \texttt{SL_rdDBus}$). As the data and address bus widths both match, no further specification is required. The challenge in generating a correct interface between the two protocols described lies in meeting the clock cycle requirements of a back to back transfer. The construction uses one predicate: $(\texttt{PLB_PAValid} \wedge (\texttt{g}(\texttt{PLB_AddrBus}), \texttt{g}(\texttt{PADDR})) = (0, 0))$ which checks if the counters corresponding to the two address buses are zero an action is performed on the signal `PLB_PAValid`. If this predicate is true, the counter is reset to $(1, 1)$. This allows for the transition between state 3 and 4 in Figure 8 to be made correctly in the interface. The interface which was generated had 31 states which reduced to 11 states after pruning was performed and timed states were identified. A 7 state FSM which was output and is shown in Figure 10. The physical characteristics of the interface are shown in Table 1. It can be seen that the interface is lightweight and does not significantly increase the amount of logic in the system.

5 Conclusion

In this paper, we have presented a general framework for modelling communication protocols, detecting when they mismatch and generating synthesizable interfaces between them if they do. The experimental results demonstrate that our framework is both easily adaptable to existing specification techniques and highly applicable to practical instances of the mismatching protocol problem. Specifically, the framework allows for detailed modelling of data buses, generation of provably correct interfaces and a new method for checking protocol compatibility. We extend the existing work in this area of research by addressing the issues of data type and bus width mismatching, transactions with multiple transfers, and provide solutions for the same.

Presently, the techniques presented here apply to synchronous protocols. We are exploring extensions to protocols operating at different frequencies which would allow for automatic synthesis of bridges between high and low speed buses. The definition of matching is quite strict as

both communciating agents are required to be equally powerful. A pair of agents with a master which is capable of interacting with many protocols and a slave which is just one of them would be declared mismatching. We plan to investigate how relaxing the requirements might allow for better identification of matching protocols.

References

[1] Gaurav Aggarwal and Daniel Gajski. Modelling guidelines for asic reuse. *Technical Report UCI-ICS-98-03*, March 1998.

[2] ARM. Amba specification. *http://www.arm.com/armtech/AMBA_spec.*

[3] Reinaldo A. Bergamaschi and William R. Lee. Designing systems-on-chip using cores. *Proceedings of the 37th Design Automation Conference*, June 2000.

[4] Gaetano Borriello and Randy H. Katz. Synthesis and optimization of interface transducer logic. *Proceedings of the International Conference of Computer Aided Design*, November 1987.

[5] Gaetano Borriello, Luciano Lavagno, and Ross B. Ortega. Interface synthesis: a vertical slice from digital logic to software components. *Proceedings of the International Conference on Computer-Aided Design*, November 1998.

[6] Kenneth L. Calvert and Simon S. Lam. Formal methods for protocol conversion. *IEE Journal on Selected Areas in Communications*, January 1990.

[7] E. R. Gansner and S. C. North. An open graph visualisation system and its applications to software engineering. *Journal of Software Practise and Experience*, May 1999.

[8] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[9] http://www.vis.org. Virtual socket interface alliance.

[10] IBM. 32-bit processor local bus, architecture specifications. *http://www-3.ibm.com/chips/products/coreconnect/.*

[11] Nancy Lynch and Frits Vaandrager. Foward and backward simulations part ii : Timing-based systems. *Journal of Information and Computation*, September 1995.

[12] Robin Milner. *Communication and Concurrency.* Prentice Hall International, London, 1989.

[13] Taken out for blind review.

[14] S. Narayan and D. D. Gajski. Interfacing incompatible protocols using interface process generation. *Proceedings of the 32nd Design Automation Conference*, June 1995.

[15] Kaoru Okumura. A formal protocol conversion method. *Proceedings of the ACM SIGCOMM*, 1986.

[16] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. *Proceedings of the International Conference on Computer-Aided Design*, November 2002.

[17] Roberto Passerone, James A. Rowson, and Alberto Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. *Proceedings of the 35th Design Automation Conference*, June 1998.

[18] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interface-based design. *Proceedings of the 34th Design Automation Conference*, June 1997.

[19] J. Smith and G. De Micheli. Automated composition of hardware components. *Proceedings of the 35th Design Automation Conference*, June 1998.

[20] Zhongping Tao, Gregor v. Bochmann, and Rachida Dssouli. A formal method for synthesizing optimized protocol converters and its application to mobile data networks. *Mobile Networks and Applications*, 1997.