Automated Interface Synthesis

Vijay D'Silva Arcot Sowmya
School of Computer Science and Engineering
University of New South Wales, Sydney, NSW 2052, Australia
E-mail: {vijayd,sowmya}@cse.unsw.edu.au

S. Ramesh Department of Computer Science and Engineering
Indian Institute of Technology Powai, Bombay 400 076
E-mail: ramesh@cse.iitb.ac.in

UNSW-CSE-TR-0325
September 2003

THE UNIVERSITY OF
NEW SOUTH WALES

Abstract

System-on-Chip (SoC) design methodologies rely heavily 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 which automates the generation of HDL descriptions of interfaces between mismatched IP communication protocols. We significantly improve and extend existing work by formalising the problem and providing a solution which addresses data mismatches, pipelining and differences in clock speeds. Importantly, the use of a formal framework enables us to generate solutions which are provably correct. The developed algorithms have been implemented and the tool used to synthesise wrappers and bridges for many SoC protocols. In particular we present a case study of the application of our algorithm to a specific design obtained from industry.

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. Approaches adopted to facilitate plug-and-play style IP reuse include the development of a few standard on-chip bus architectures such as CoreConnect[8] from IBM, AMBA[2] from ARM among others and the work of the VSI Alliance[1] and the OCP-IP[9] consortium. Figure 1 shows a typical bus-based SoC architecture. When IP blocks and buses use different communication protocols or operate at different speeds, wrappers and bridges are introduced as shown. Unfortunately, the vision of assembling SoCs using 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 a SoC is largely a manual process requiring considerable design effort. If possible, protocol mismatches are resolved using off-the-shelf wrappers. This solution has the side effect of increasing verification overheads.
- Verification of the entire system is a bottle neck due to interface and timing issues.

Interface synthesis is an area of research[5, 14] that seeks to automate the process of interconnecting components at different levels. Interfaces at low levels are circuits or devices primarily concerned with physical quantities such as voltage and capacitance while high level interfaces are state machines or programs which address abstract behaviours such as the interaction between state machines or processes. We focus on the synthesis of high level interfaces which facilitate communication between possibly mismatched protocols described as Finite State Machines(FSMs).

1.2 Related Work

The problem of automatically resolving protocol mismatches by synthesising interfaces has been addressed in the literature[14]. Differences in signalling convention, data width and type, sequencing of data, and clock speeds have been identified as the causes of protocol mismatch.

Preliminary work in this area was done by Borriello and Katz[4] who used timing diagram specifications of protocols to construct event graphs and generate a transducer circuit. The designer must provide information for the correct merging of event graphs and data buses must have the same names.

Narayan and Gajski[10] decompose a sequential protocol specification into combinations of five basic operations, and organise the protocol behaviour as totally ordered sets of guarded executions. The sets transferring the *same* amount of data are matched and an interface is constructed.

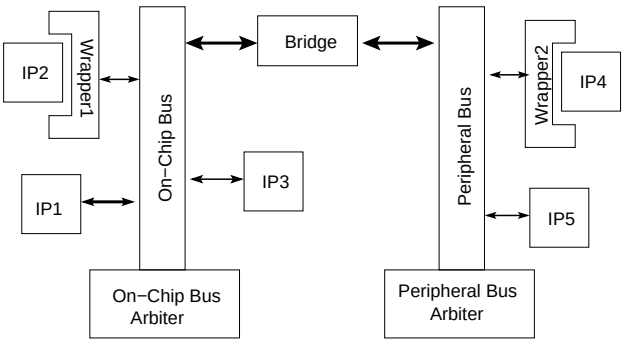


Figure 1: System-On-Chip Bus Architecture

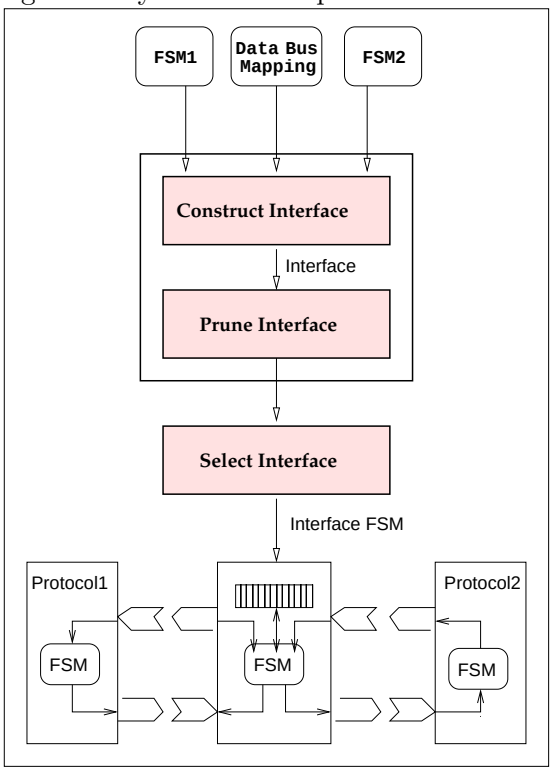


Figure 2: Design Flow for Interface Synthesis

Smith and De Micheli[16] map any given protocol into a standard communication scheme which requires that a protocol be either a sender or a receiver. Their scheme can be applied in a multi-party communication environment but their solution is quite expensive as there is a six cycle latency between a data read and write and an internal arbiter is used that significantly increases the amount of logic in the system. This work was extended by Shin and Gajski[15] by using protocol flow graph specifications to synthesise interfaces which use queues and internal control logic to regulate buffering.

Passerone et al.[13] provide an interface synthesis algorithm for mismatched synchronous protocols specified as regular expressions. Their technique cannot be easily extended to different kinds of data and clock speed mismatches. Recently, Passerone et al.[12] stated that the above methods lack a mathematically sound formalisation and attempted a game theoretic formalisation. The synthesis procedure is illustrated with an example that handles reordering of data. No algorithm is presented, so it is unclear how the proposed technique can be applied to any two arbitrary protocols.

Interface synthesis has been addressed in the context of network protocols as the protocol conversion problem[6, 11, 17]. Due to the relative complexity of network protocols, these solutions use a high level of abstraction and require complex specifications to be provided. Further, the systems considered are asynchronous unlike most on-chip protocols.

None of the solutions provided in the hardware literature apply to protocols with arbitrary branching structures and loops. Data mismatches are partially addressed while complex behaviours such as pipelining are not and current solutions for clock speed mismatches result in large and complex interfaces.

1.3 Contributions and Overview

In this report, we present a lightweight formalisation of the protocol mismatch problem and derive an algorithm for interface synthesis. The framework assumes that the protocols are modelled as FSMs and facilitates design space exploration. We identify the conditions under which two protocols will mismatch. This aspect of the formalisation is novel and has not been undertaken in the literature. Special attention is provided to the modelling of data, an issue which existing solutions do not address. We present an algorithm which uses an extremely simple specification to automate the construction of interfaces capable of handling protocols with mismatched data widths, types, clock speeds and pipelining behaviour. In addition, we use the notion of matched protocols to identify a correctness criterion and show that the interface constructed is correct.

The modelling and synthesis flow is shown in Figure 2. The designer should provide FSM style protocol descriptions and a mapping between the data buses of the protocols. We use FSMs as they capture all the required behaviour and simultaneously minimise the designer's effort. The FSM descriptions are used to synthesise all possible correct interfaces between the two protocols. The designers may then select the interface best suited to their requirements and it can then be translated into HDL code. We have applied our technique to synthesise wrappers and bridges between commonly used SoC protocols such as the Coreconnect Processor Local Bus, AMBA Advanced System Bus, AMBA Advanced High-performance Bus and the OCPIP Open Core Protocol.

The report is organised as follows. Section 2 introduces the preliminary definitions

and formalises the notions of protocol matches and mismatches. Section 3 contains a formal description of the interface, the interface synthesis algorithm and the extension to mismatched clock speeds. Section 4 presents the experimental results and we conclude in Section 5.

2 Protocol Specification and Modelling

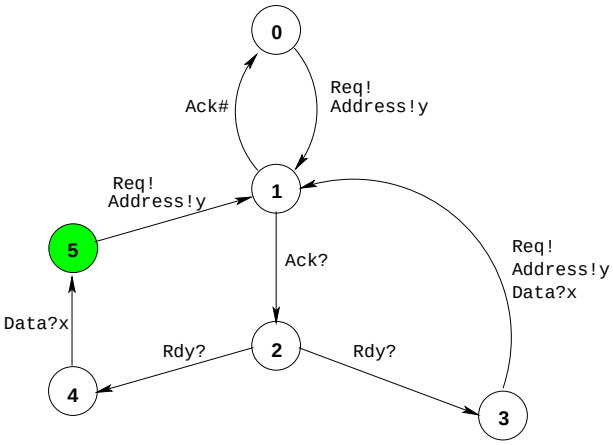
We illustrate our approach using a simple protocol referred to as *Pipeline*, which supports pipelining. The protocol requests a transfer using the signal `Req` and writes to the address bus `Addr`. This operation is repeated until an acknowledgement `Ack` is received. The protocol then waits for a signal `Rdy` to be asserted, confirming that data will be written to the bus. If a single read is to be performed, the protocol will read from the data bus `Data` and complete the transaction. If a sequence of reads is desired, the protocol will pipeline the address and data phases by sending the next request and address and reading data from the bus in the same transition. When it completes a finite sequence of transfers, the protocol will stop until it is ready to start a fresh transaction. All actions performed by the protocol are synchronised with a clock which emits ticks at regular intervals.

The behaviour of *Pipeline* can be modelled as a FSM shown in Figure 3(a). The state machine has a set of input channels `Ack, Data, Rdy` and output channels `Req, Address`. `Data` and `Address` are buses or data channels and all the others are control channels. *Pipeline* begins its operation in state 0 and transits to state 1 performing the write operation `Req!` which causes an event to take place on the channel `Req`. In state 1, if the acknowledgement signal `Ack` does not arrive, it is detected by the operation `Ack#` and a transition is made to state 0 in order to resend the address. In state 2, when the input `Rdy` occurs, denoted by `Rdy?`, the protocol will transit to either state 3 or 4, depending upon an internal condition that indicates whether a non-pipelined or a pipelined transfer is to be performed. The details of the internal condition are irrelevant at this level of description. If a non-pipelined operation is desired, the protocol will reach state 4, read data and transit to state 5 which is the final state. If a protocol is in a final state, we can infer that one or more data transfers have occurred. In all other respects, the final state is identical to the initial state.

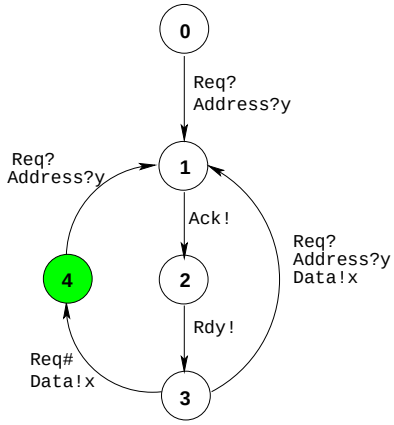
The sequence of transitions from state 0 through states 1,2 and 4 to state 5 is an example of a *transaction run*. When the protocol completes a transaction, it resides in the final state until a new transaction is to be undertaken.

Figure 3(b) models a slave corresponding to *Pipeline* called *PipelineSlave*. When *Pipeline* makes a request, *PipelineSlave* will receive it, read the address and first send an acknowledgement and then the `Rdy` signal to reach state 3. In state 3, *PipelineSlave* will transit to state 1 writing the data which was requested to the bus and will simultaneously read the address corresponding to the pipelined request if one is made. If there is no pipelined request, *PipelineSlave* will transit to state 4 which is the final state. It can be seen from this behaviour that *Pipeline* and *PipelineSlave* can communicate successfully to complete a transaction.

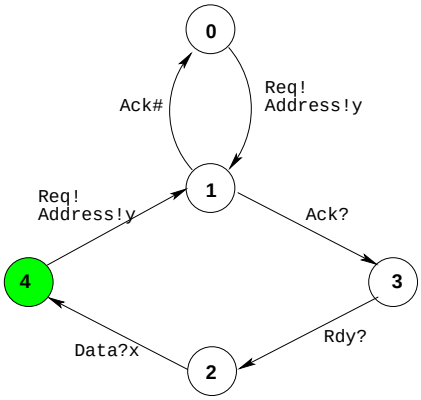
Figure 3(c) is another master, *NoPipeline*, for *PipelineSlave*. Its interaction is similar to that of *Pipeline* but it will never cause *PipelineSlave* to transit from state 3 directly



(a) *Pipeline*



(b) *PipelineSlave*



(c) *NoPipeline*

Figure 3: Examples of Protocols

to state 1 as it does not require pipelined operation. The protocol pair *NoPipeline* and *PipelineSlave* illustrate that it may not always be clear via naive visual inspection("Do these protocols *look* the same?") if two protocols match. Hence, there is a need for a systematic approach to determining whether a pair of protocols match. We attempt this through Definitions 1, 6 and 7.

Definition 1 *A protocol is a communicating Finite State Machine* $P = (Q, \Sigma, \Delta, V, \mathcal{A}, \longrightarrow, q_0, q_f)$. *$Q$ is a set of control states,* $\Sigma = \Sigma^I \cup \Sigma^O$ *and* $\Delta = \Delta^I \cup \Delta^O$ *are sets of disjoint input and output control and data channels. V is a set of internal variables and $\mathcal{A}$ is the set of actions that can be performed.* $\longrightarrow \subseteq Q \times \mathcal{A} \times Q$ *is the state transition relation.* q_0 *and* q_f *are the initial and final states.*

An action S in $\mathcal{A}$ is of the form $G : Nb$ where G is a set of guards or blocking operations and Nb is a set of nonblocking operations which can be performed. Guards check for the presence, denoted as $c?$, or absence, denoted as $c\#$, of events on control channel c. Nonblocking operations are writes on control channels , denoted as $c!$ which cause events to occur, and reads or writes on data channels, denoted as $d!, d?x$. τ is the empty action.

A transition is written as $q \xrightarrow{G:Nb} q'$ and a sequence of transitions as $q \stackrel{\alpha}{\Longrightarrow} q'$ where $\alpha = G_1 : Nb_1, \ldots, G_k : Nb_k$. A *transaction run* is a sequence of transitions $q_0 \stackrel{\alpha}{\Longrightarrow} q_f$ in which the protocol begins and completes a transfer or sequence of transfers.

The predicate `blocking`(q) is true in a state q if all outgoing transitions are labelled with blocking actions. q is non-blocking if all outgoing transitions have only nonblocking actions. When the protocol is in a state and a clock tick occurs, the transition whose guard evaluates to true is taken. If more than one guard is true, a nondeterministic choice is made. We require that all actions either cause events or respond to them and all states are either blocking or nonblocking.

Further we restrict our focus to a subclass of nondeterministic protocols which are *weakly deterministic*. A protocol is weakly deterministic if, when a state has multiple target states for a given action, these target states are either distinguishable from each other by their output events, or will lead to states which are distinguishable in this manner. *Pipeline* is an example of a weakly deterministic protocol. As observed earlier, state 2 is a nondeterministic state. But its two successors 3 and 4 are distinguishable as 3 emits signal `Req` while 4 does not. In fact, the slave protocol `PipelineSlave` infers that the requested transfer is pipelined by checking for the presence of the signal `Req`. As a result, even though one of the protocols is nondeterministic, together they compute unique transactions.

These notions are formalised in Definitions 2, 3 and 4 below.

Definition 2 *A action is* legal *if it does not contain a control read as well as a control write. That is, for any c, c' if $c? \in G_1 : Nb_1$ then $c'! \notin G_2 : Nb_2$.*

Legal actions ensure that causality cycles do not occur by requiring actions to either cause events or respond to them.

Definition 3 *Two non-blocking states q_1, q_2 are* output distinguishable *if for any $q_1', q_2', G_1 : Nb_1, G_2 : Nb_2 : q_1 \xrightarrow{G_1:Nb_1} q_1', q_2 \xrightarrow{G_2:Nb_2} q_2'$, there exists at least one $c! \in G_1 : Nb_1$ such that $c! \notin G_2 : Nb_2$ or vice versa.*

Definition 4 *A protocol is weakly deterministic if* $q_0 \stackrel{\alpha}{\Longrightarrow} q_1 \stackrel{G_1:Nb_1}{\longrightarrow} q_1'$ *and* $q_0 \stackrel{\alpha}{\Longrightarrow} q_2 \stackrel{G_2:Nb_2}{\longrightarrow} q_2'$ *and* $q_1 \neq q_2$ *then,* q_1 *and* q_2 *are output distinguishable.*

We encounter weak determinism when modelling protocols because only the external events are modelled. Transitions and choices which are governed by internal conditions are not visible resulting in non-determinism in the model. Henceforth, when discussing actions, we shall use S_1 as syntactic sugar for the action $G_1 : Nb_1$.

2.1 Protocol Compatibility

We now focus on the problem of protocol mismatch. Consider the protocol *Handshake* in Figure 5(a). When selected for a read transfer by the presence of events on the channels `SEL` and `READ` it will read the address from `ADDR`. If the signal `ENABLE` arrives, it will write data on the data channel `RData` and transit to state 3 completing the transfer.

Pipeline (which has been replicated in Figure 5(c) for readability) and *Handshake* are a pair of protocols which are mismatched for many reasons: they have different channel names, different number of operations on channels and *Pipeline* supports pipelined operation while *Handshake* does not.

Intuitively, we would like to declare a pair of protocols as matched if they have the same number and names of channels. Further, if we 'run' the two protocols together, they should enable each other to exchange a series of messages, make progress and terminate the transaction successfully. While they run concurrently, any control or data signal sent by one should be received by the other. If one protocol is blocking on a set of guards, an action by the other should make at least one of the guards true. Conversely, if one protocol performs a nonblocking action, the other protocol should respond to it.

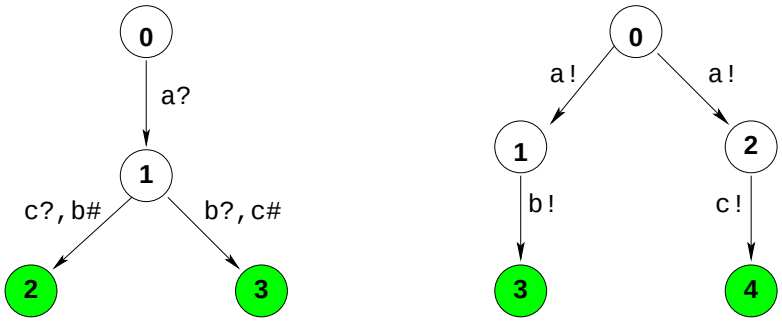
To formalise the notions above, we define a predicate `permit`.

Definition 5 *Given a pair of actions* S_1 *and* S_2, $\texttt{permit}(S_1, S_2)$ *holds whenever if* S_1 *contains a read operation* $c?$ *then* S_2 *contains the write operation* $c!$ *and vice versa. Further if one has the operation* $c\#$ *then the other should not have* $c!$.

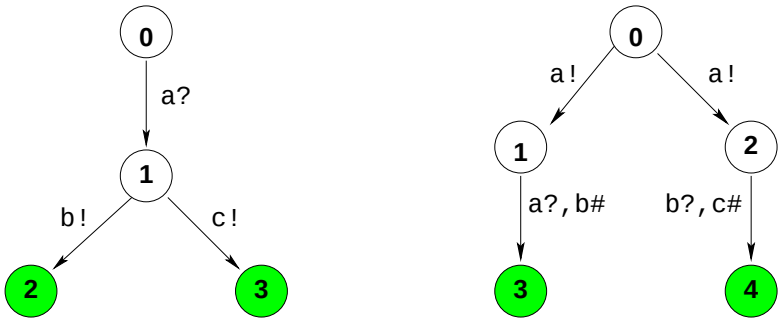
We now define a relation between the states Q_1 and Q_2 of two protocols P_1 and P_2 with final states r_f and t_f that formalise the notion of a protocol match.

Definition 6 *A transaction relation is a symmetric binary relation* $\mathcal{R} \subseteq Q_1 \times Q_2$ *satisfying:*

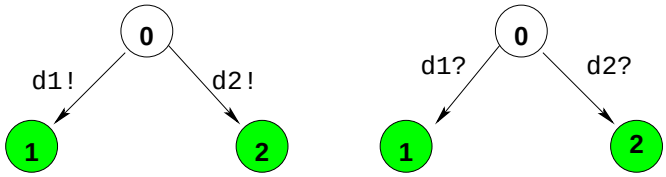
1. $\langle r_f, t_f \rangle \in \mathcal{R}$
2. *if* $\langle r, t \rangle \in \mathcal{R}$ *and* $\neg\texttt{blocking}(\texttt{r})$ *and* $\neg\texttt{blocking}(t)$ *then, whenever* $r \stackrel{S_1}{\longrightarrow} r'$ *and* $t \stackrel{S_2}{\longrightarrow} t'$ *it holds that* $\texttt{permit}(S_1, S_2)$ *and* $\langle r', t' \rangle \in R$
3. *if* $\langle r, t \rangle \in \mathcal{R}$ *and* $\neg\texttt{blocking}(r)$ *and* $\texttt{blocking}(t)$ *then, whenever* $r \stackrel{S_1}{\longrightarrow} r'$ *it holds that there exists* $S_2, t' : (t \stackrel{S_2}{\longrightarrow} t'$ *and* $\texttt{permit}(S_1, S_2))$ *and for all such* $S_2, t' : \langle r', t' \rangle \in R$



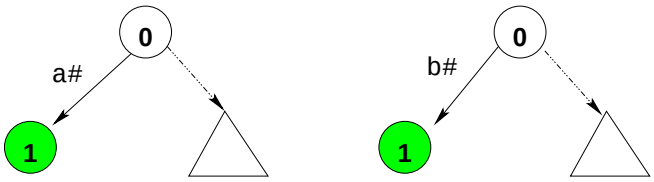
(a) *Matched Pair with weak determinism*



(b) *Unmatched Pair*

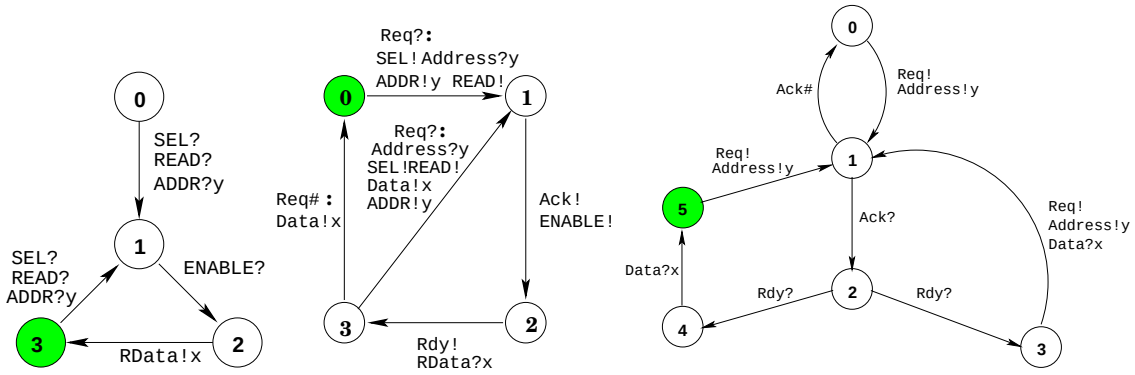


(c) *Unmatched Pair with data*



(d) *Matched pair with blocking states*

Figure 4: Protocol pairs



(a) *Handshake*

(b) Interface for *Handshake* and *Pipeline*

(c) *Pipeline*

Figure 5: Protocols and Interface

4. *if* $\langle r, t\rangle \in \mathcal{R}$ *and* $\texttt{blocking}(r)$ *and* $\texttt{blocking}(t)$ *then, whenever* $r \xrightarrow{S_1} r'$ *and* $t \xrightarrow{S_2} t'$ *such that* $\texttt{permit}(S_1, S_2)$ *it holds that* $\langle r', t'\rangle \in R$

Using this relation, we formalise the notion of matched protocol pairs.

Definition 7 *A protocol pair* P_1 *and* P_2 *with initial states* r_0 *and* t_0 *is said to match if there exists a transaction relation* $\mathcal{R}$ *such that* $\langle r_0, t_0\rangle \in \mathcal{R}$.

Figure 4 illustrates a few cases of protocol match and mismatch which motivate the conditions in the transaction relation. Figure 4(c) illustrates two non blocking protocols. Their behaviour is unacceptable and is rejected by the second condition as one protocol might write data to the channel d_1 and the other might attempt to read from d_2.

Figures 4(a) and 4(b) are protocol pairs in which one protocol is weakly deterministic. The protocols in Figure 4(b) do not match as after a signal is sent on channel a one protocol may send a signal on either channel b or channel c while the other protocol may be capable of receiving only one depending on whether it is in state 1 or state 2.

Figure 4(d) illustrates a protocol pair which can reach their final states if no signal is sent on channels a and b. This is allowed by the last condition of the relation. Though protocols of the form shown in Figures 4(c) and 4(d) are quite unlikely to occur in real designs, they have been included as they motivate conditions which have been included for theoretical correctness.

The transaction relation for *Pipeline* and *PipelineSlave* is $\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle, \langle 4,3\rangle, \langle 5,4\rangle\}$. No such transaction relation exists between the protocols *Pipeline* and *Handshake* and we may conclude that the two protocols *mismatch*.

3 Interface Synthesis

3.1 Interfaces between Protocols

In the case of a protocol mismatch, system designers will introduce a device called an *interface* which mediates between the protocols as shown in Figure 2. This is akin to wrappers and bridges introduced by designers when interconnecting IP blocks. The interface will be connected to the channels of both protocols and can distinguish between them. It will be two-phased, receiving data and control from one and sending to the other. Further, it will contain counters and data buffers and will ensure that all data which is read is output and no new data is independently generated by the interface.

Figure 5(b) describes the behaviour of an interface between the protocols *Handshake* and *Pipeline*. It assumes that `Address,ADDR` and `RData,Data` are corresponding pairs of channels of equal width. When the interface is in state 0 and receives a request from *Pipeline* it will read data from `Address`, write the same information to `ADDR` and write to the channel `Sel` causing both protocols to transit from their initial states. In state 1, the interface will write to the channels `Ack` and `ENABLE` and cause both protocols to transit to their respective state 2. State 3 handles the interesting case of *Pipeline* requesting a pipelined read operation. The data which has been obtained from *Handshake* is sent to *Pipeline* and the address obtained from *Pipeline* is also sent to *Handshake* which is selected for a new transfer. This transition from state 3 to 1 preserves the latency which a pipelined operation attempts to achieve. The interface will decide if a pipelined operation is being requested by checking for the presence or absence of `Req` in state 3.

The interface in Figure 5(b) can be **automatically synthesised** from the FSM descriptions of *Pipeline* and *Handshake*. In general, corresponding channels may have different data types and widths. The interface will then have to perform type transformations and maintain counters to regulate the buffering of data. The next section describes the synthesis procedure.

3.2 Interface Synthesis Algorithm

Consider two protocols $P_1 = (Q_1, \Sigma_1, \Delta_1, V_1, \mathcal{A}_1, \longrightarrow, r_0, r_f)$ and $P_2 = (Q_2, \Sigma_2, \Delta_2, V_2, \mathcal{A}_2, \longrightarrow, t_0, t_f)$ which are mismatched. In order to synthesise an interface, a specification $f : \Delta_1 \rightarrow \Delta_2$ relating their data channels must be provided. The specification will indicate the intended interconnections between data channels. We assume that each data channel will only be connected to one other. If there is a data type mismatch between any such pair of channels $d, f(d)$, a type transformer $\mathcal{T} : d.T \rightarrow f(d).T$ may be specified.

The interface $I = (Q, \Sigma, \Delta, V, \mathcal{A}, \longrightarrow, q_0, q_f)$ is a FSM constructed by Algorithm 1 where

- $Q \subseteq \{\langle Q_R, Q_T \rangle | Q_R \subseteq Q_1, Q_T \subseteq Q_2\}$. Each control state of I corresponds to a set of states of P_1 and a set of states from P_2. This is necessitated by the weak determinism in P_1 and P_2.
- $\Sigma^I = \Sigma_1^O \cup \Sigma_2^O$, $\Sigma^O = \Sigma_1^I \cup \Sigma_2^I$. I will receive input from the output channels of P_1 and P_2 and will write to the input channels of P_1 and P_2.

- $\Delta^I = \Delta_1^O \cup \Delta_2^O$, $\Delta^O = \Delta_1^I \cup \Delta_2^I$.
- V is a set of data buffers such that there is one buffer y_i for each pair of data channels $d_i, f(d_i)$.
- $\mathcal{A}$ is a set containing one action complementary to each of the actions in $\mathcal{A}_1$ and $\mathcal{A}_2$.
- $\longrightarrow$ relates the different states in Q as per the algorithm.
- $q_0 = \langle\{r_0\}, \{t_0\}\rangle$ is the initial control state and $q_f = \langle\{r_f\}, \{t_f\}\rangle$ is the final state.

In addition, I will have a set of counters $X = \{x_i\}$ with one counter for each pair of data channels. I will communicate by exchanging messages with P_1 and P_2. Each counter in X corresponds to a pair of related data channels and is incremented when data is received on one and decremented when it is written out on the other. I can use the value of a counter to determine if a buffer contains data that has not been written out.

The transition relation $\longrightarrow$ is derived from the transition relations of P_1 and P_2 as can be seen in Algorithm 1. For every reachable pair of sets of states of P_1 and P_2, the interface should be able to detect and respond to any action performed by the two protocols. If a data channel say d_i is read from, the data is stored in buffer y_i and the counter x_i will be incremented as required. When data has to be written to the channel $f(d_i)$ that corresponds to d_i, the specified transformation $\mathcal{T}$ is applied to y_i, the type transformed data is written to $f(d_i)$ and x_i is decremented. Using the counters, the interface determines the validity of data being written to a bus. A complete description of the control state of the interface and its counters in that state is written as $[\langle R, T\rangle, X]$ where R is a set of states from P_1, T a set of states from P_2 and X is the status of the counters.

Algorithm 1 formalises these intuitive ideas. Given a set of states of P_1 and P_2 the body of the *for* loop computes all possible actions which can be performed by the interface using the predicate $valid(S, X)$ defined in Algorithm 4. $valid(S, X)$ is true if, for every data write operation in a candidate action S, all required data has been read or will be read in S. The states that will be reached by performing *valid* actions are computed using Algorithm 2 and the counters are updated as shown in Algorithm 3. The definition of $valid(S, X)$ can be augmented with any resource constraints that are to be imposed on the interface such as bounds on buffer sizes.

Algorithm 1 algorithmic $Q = \emptyset$ Q is the state space of I `pendingStates` $= \{[\{r_0\}, \{t_0\}, X]\}$ Let $[R, T, X]$ be some state in the set `pendingStates` `pendingState` $\neq \emptyset$ $r \in R, S_1 : r \xrightarrow{S_1}, t \in T, S_2 : t \xrightarrow{S_2} S_1' :=$ ComputeComplement(S_1) $S_2 :=$ ComputeComplement(S_2') $valid(S_1' \cup S_2', X)$ $R' :=$ ComputeTarget(R, S_1) $T' :=$ ComputeTarget(T, S_2) $X' :=$ ModifyCounters$(X, S_1' \cup S_2')$ AddTransition : $[R, T, X] \xrightarrow{S_1' \cup S_2'} [R', T', X']$ $[R', T', X'] \notin Q \cup$ `pendingStates` Add $[R', T', X']$ to `pendingStates` Add $[R, T, X]$ to Q and remove from `pendingStates` Prune()

InterfaceSynthesis(P_1, P_2)

Algorithm 2 algorithmic $Target = \emptyset$ $q \in Source$ $Target := Target \cup \{q' | q \xrightarrow{S} q'\}$ return $Target$

ComputeTarget(Source,S)

Algorithm 3 algorithmic $d_i?y, d_j! \in S$ $incr(x_i)$, $decr(x_j)$ return X

ModifyCounters(X,S)

The interface that is generated may contain *bad* or *dead* states. The algorithm only ensures that the actions performed by the interface are valid. On the other hand, actions that the interface should perform in a certain state might not be present because they may not be valid in that state. Such states are eliminated by the second *for* loop of the pruning procedure in Algorithm 5. The pruning procedure also analyses the loops in the interface and removes loops in which a counter value increases, as such behaviour will lead to interfaces which require unbounded buffers. This operation is performed by the first for loop in Algorithm 5. If the pruning procedure produces an interface that contains no states, we can infer that no interface exists for that pair of protocols.

Algorithm 4 algorithmic $d_i?y \notin S$ for any d_i $\forall d_j!S : x_i \geq 1$ return `true` $S_w = \{d_i?y | d_i?y \in S\}$ $X' :=$ ModifyCounters(X, S) return $valid(S, X')$

$valid(S, X)$

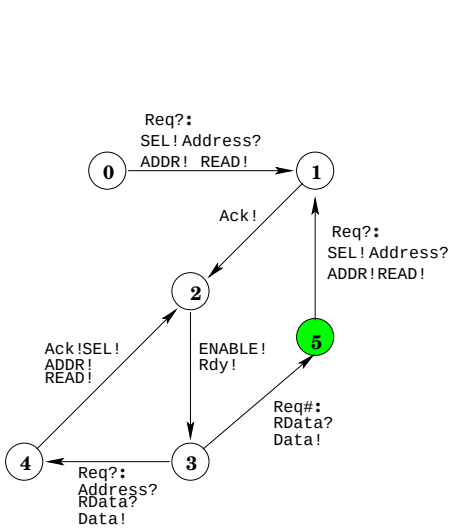
Algorithm 5 algorithmic Loops : $[R, T, X] \stackrel{\alpha}{\Longrightarrow} [R_1, T_1, X_1] \stackrel{S}{\longrightarrow} [R, T, X']$ $x_i \in X$ and $x'_i \in X' : x'_i > x_i$ remove transition :$[R_1, T_1, X_1] \stackrel{S}{\longrightarrow} [R, T, X']$ $[R, T, X] \in Q$ ($r \in R$ and $\neg$`blocking`(r) and $r \stackrel{S}{\longrightarrow} and [R, T, X] \stackrel{S}{\nrightarrow}$)
or ($t \in T$ and $\neg$`blocking`(t) and $t \stackrel{S'}{\longrightarrow}$ and $[R, T, X] \stackrel{S'}{\nrightarrow}$) remove $[R, T, X]$ from Q no states are removed

Prune()

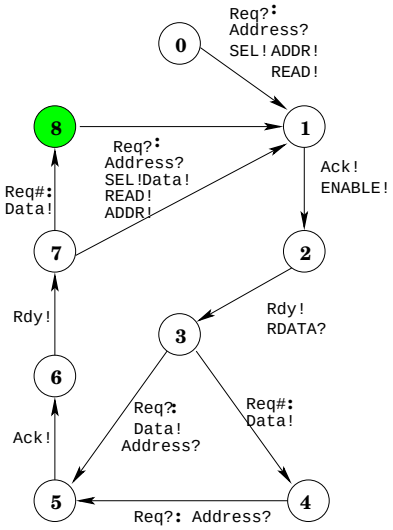
After pruning, the interface generated might still contain redundant paths. The designer could eliminate some of these paths depending on the optimisation to be performed.

For example, the interfaces in Figures 6(a) and 5(b) can both be obtained from the result of the algorithm when applied to the mismatched protocol pair *Pipeline* and *Handshake* with the specification `RData` → `Data`, if their data buses have the same width. The interface in Figure 6(a) differs from Figure 5(b) by sending the `ENABLE` signal to *Handshake* in state 2 instead of in state 1. As a result, when data is read from `RData` it is written to `Data` simultaneously as can be seen in the transitions from state 3 to states 4 and 5. In such a situation, a buffer will not be required for the channel pair `RData,Data`. Though pipelined operation is supported in the transitions from state 3 to 4 to 2, it has a greater latency as a request cannot be sent to *Handshake* in the same transition in which a pipelined request is received. So, while the interface in Figure 5(b) has a lower latency for each transfer, Figure 6(a) does not require a buffer for data.

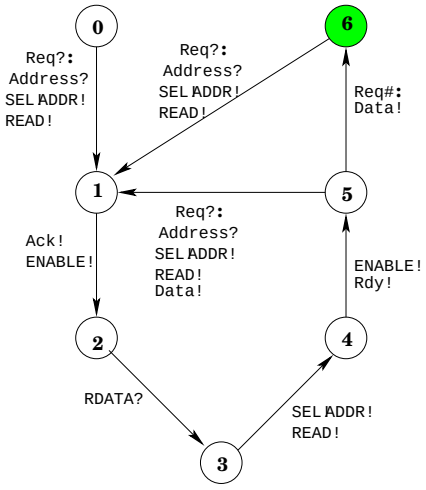
Figure 6(b) is an interface synthesised when `RData` is twice as wide as `Data`. The operation of converting one packet from `RData` to two in `Data` has been abstracted out. It can be seen that for all cycles within the interface `RData` is read from only once and `Data` is written to twice as required. Data is written to *Pipeline* in the outgoing transitions



(a) Matching data widths



(b) `RData` twice as wide as `Data`



(c) `Data` twice as wide as `RData`

Figure 6: Automatically synthesised Interfaces for *Handshake* and *Pipeline*

from states 3 and 7. Both these states have two outgoing transitions each as a pipelined transfer might be requested. Note that a request is sent to *Handshake* only in response to the second pipelined request in the transition from state 7 to 1. In the first pipelined request in the transition from state 3 to 5 there is no interaction with *Handshake* as all the required data has already been obtained.

Figure 6(c) illustrates an interface for the situation when `Data` is twice as wide as `RData`. Once again, the conversion of two packets of data from `RData` to one packet has been abstracted out. It can also be seen that in every transaction run, a read from `RData` will occur twice as often as a write to `Data`. In contrast to Figure 6(b), only state 5 has more than one outgoing transition as the interface drives *Pipeline* to a state in which a pipelined request can be made only once during each transfer. In the transition from state 5 to state 1 the interface receives a pipelined request from *Pipeline* and sends a request to *Handshake*, which does not affect the latency of pipelined requests.

All these interfaces were constructed from the same algorithm. Note that the states of Figure 6(b) and 6(c) are extended states which include the counter values.

3.3 Multiple clock speeds

We illustrate with an example how the interface synthesis algorithm can be extended to protocols which operate on different clocks. Suppose *Pipeline* runs at $\texttt{clk}_1$ and *Handshake* at $\texttt{clk}_2$ and $\texttt{clk}_2$ is twice as fast as $\texttt{clk}_1$. First we construct a clock FSM as shown in Figure 7(a). Note that $\texttt{clk}_2$ appears twice as often as $\texttt{clk}_1$. The clock FSM and *Pipeline* can be run synchronously and this behaviour can be modelled as a new FSM shown in Figure 7(b). Note that the new machine has τ transitions which correspond to the transitions in the clock FSM in which $\texttt{clk}_1$ does not occur. This technique, which is quite standard in the synchronous language literature[7], is called *oversampling*.

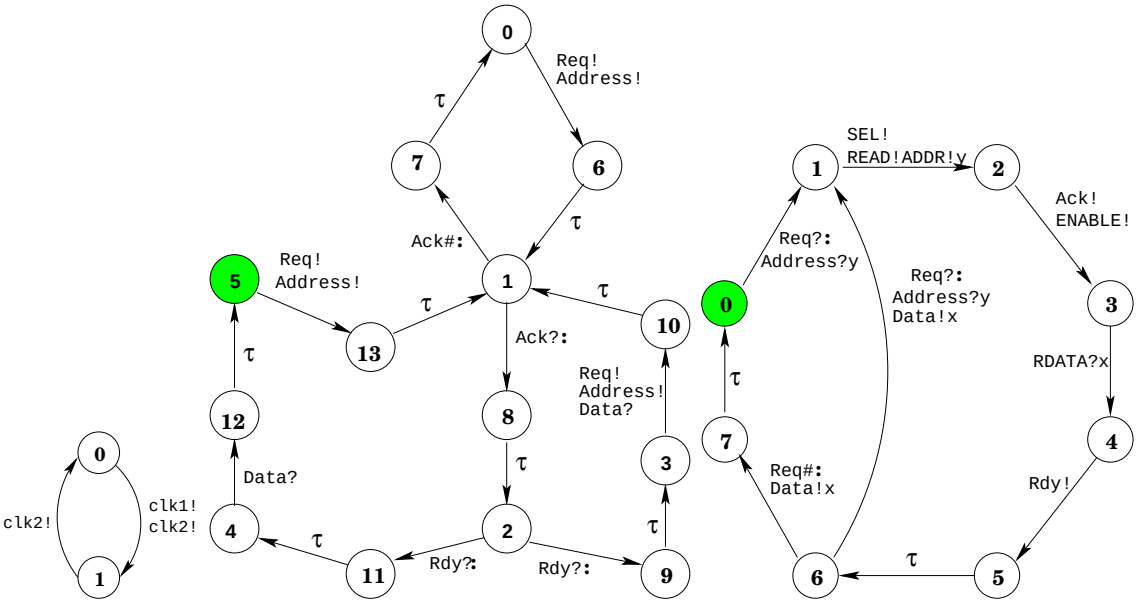
Similarly, using the same clock FSM, a modified FSM for *Handshake* is constructed. Now, we can apply the algorithm to these modified FSMs and obtain a new interface shown in Figure 7(c)

Further, it is not required that the two clock speeds be multiples of each other as in the example because the relative speeds of any two clocks with respect to each other can always be expressed using the clock FSM. Figure 7(d) describes the FSM *Clock* when the clock speeds of *Pipeline* and *Handshake* have the ratio 2:3.

3.4 Correctness of Algorithm

As mentioned in Section 3, the purpose of constructing an interface for a pair of mismatched protocols P_1 and P_2 is to modify one of them, say P_2, to P_2' so that P_1 and P_2' are a matched protocol pair.

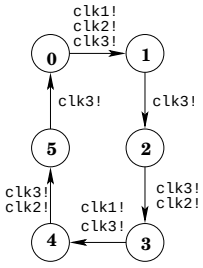
The operation of modifying a protocol with an interface can be modelled using the synchronous parallel composition operator $||$. Figure 8 is the result of the parallel composition of the interface in Figure 8 with *Handshake*. This composition performs a synchronous product of the two FSMs and hides all communication between the two FSMs and is performed using a parallel composition operator $||$, the details of which are provided in



(a) FSM:Clock

(b) Combined FSM of Clock and Pipeline

(c) Interface with clock $\texttt{clk}_2$



(d) Clock for ratio 2:3

Figure 7: Clock FSM and Interface

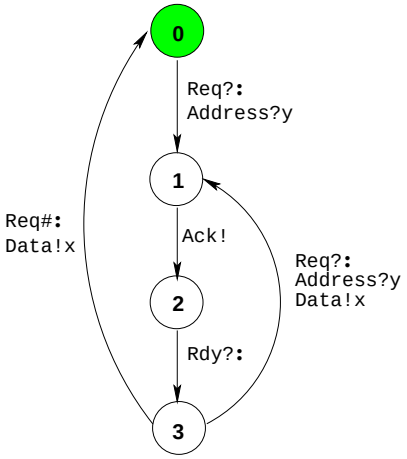


Figure 8: Composition of Interface and Handshake

Definition 8. It can be observed that *Pipeline* and (*Interface* || *Handshake*) are a matched protocol pair.

Definition 8 *The result of a parallel composition* $P_1||P_2$ *of two protocols* P_1 *and* P_2 *with a shared set of channels* $C = (\Sigma_1 \cap \Sigma_2) \cup (\Delta_1 \cap \Delta_2)$ *is a FSM defined by*

- $Q_{P_1||P_2} = Q_1 \times Q_2$
- $\Sigma^I_{P_1||P_2} = (\Sigma^I_1 \cup \Sigma^I_2)\backslash C$, $\Sigma^O_{P_1||P_2} = (\Sigma^O_1 \cup \Sigma^O_2)\backslash C$
- $\Delta_{P_1||P_2} = (\Delta_1 \cup \Delta_2)\backslash C$
- $\mathcal{A}_{P_1||P_2} = (\mathcal{A}_1 \cup \mathcal{A}_2)\backslash \mathcal{A}_C$
- $\langle r_0, s_0 \rangle$ *is the initial state and* $\langle r_f, s_f \rangle$ *is the final state*
- $\longrightarrow_{P_1||P_2}$*: given* $r \xrightarrow{S} r'$ *and* $s \xrightarrow{S'} s'$ *and* $S_c = (S \cup S')\backslash \mathcal{A}_C$ *the following rule defines* $\longrightarrow_{P_1||P_2}$

$$\frac{r \xrightarrow{S_1} r' \wedge t \xrightarrow{S_2} t' \wedge S = (S_1 \cup S_2)\backslash \mathcal{A}_C}{\langle r, t \rangle \xrightarrow{S} \langle r', t' \rangle}$$

Theorem 1 states that the algorithm to construct I is correct.

Theorem 1 *Given two mismatched protocols* P_1 *and* P_2*, Algorithm 1 will generate an interface* I *with the property that* $P_1, (I||P_2)$ *are a matched protocol pair.*

One could equivalently state that $(P_1||I)$ and P_2 match.
Proof Sketch: We first consider the product $I||P_2$. The following observations will be utilised in the proof.

Protocol	Pipelining	Errors	Burst	Abort
Advanced System Bus	√			
Processor Local Bus	√		√	
Advanced High-performance Bus	√	√	√	
Open Core Protocol	√	√	√	√

Table 1: Features of Protocols Considered

1. For all states $[\langle R, T\rangle, t]$ in the product machine, $t \in T$.

2. If the property $\texttt{permit}(S_1, S_2)$ holds, a blocking operation in S_1 will have a non-blocking complementary operation in S_2 and vice versa.

3. If P_1 and P_2 are weakly deterministic, then in every state $\langle R, T\rangle$ of the interface, either $\bigwedge_{r\in R} blocking(r)$ or $\bigwedge_{r\in R} \neg blocking(r)$ is true.

Construct the relation $\langle r, [\langle R, T\rangle, t]\rangle \in \mathcal{R}$ if $r \in R$. It remains to show that $\mathcal{R}$ is a transaction relation. Consider the following cases:

1. $blocking(r)$: We have that $\neg blocking([\langle R, T\rangle, t]$(Observation 2,3). An outgoing transition is guaranteed to exist as: if $blocking(t)$ then $[\langle R, T\rangle, t]$ will have at least one outgoing transition as complete monomials are considered. If $\neg blocking(t)$ then $\langle R, T\rangle$ will have all the required outgoing transitions because all states which are otherwise were eliminated in the pruning step. So, at least one outgoing transition is guaranteed to exist. As transitions from blocking states are labelled with complete monomials, there will be an outgoing transition from r which this will match.

2. $\neg blocking(r)$: We have that $blocking([\langle R, T\rangle, t])$(Observation 2,3). As a result of pruning, we know that if a state exists with $r \in R$ then the conditions required by the transaction relation will be met.

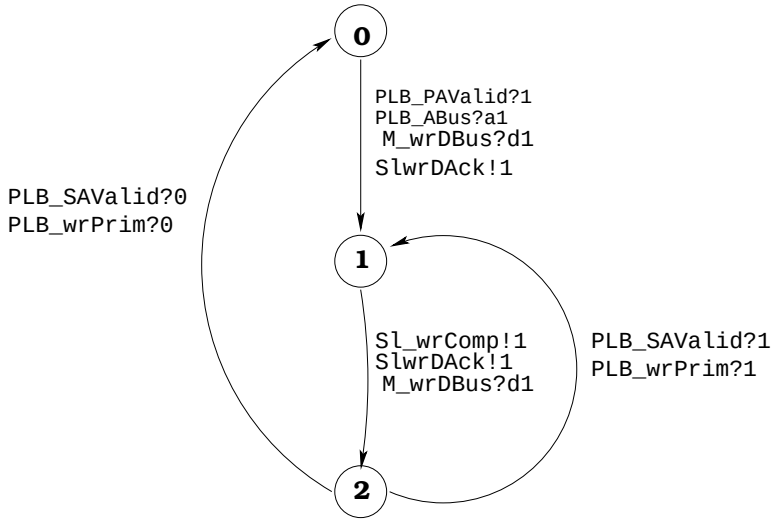
3. Finally if $r \xrightarrow{S} r'$ there exists a transition $[\langle R, T\rangle, t] \xrightarrow{S'} [\langle R', T'\rangle, t']$ for which it holds that $r' \in R'$ and $\texttt{permit}(S, S')$ as a result of Algorithm 1. We have $\langle r', [\langle R', T'\rangle, t']\rangle \in \mathcal{R}$ as required.

4 Experimental Results

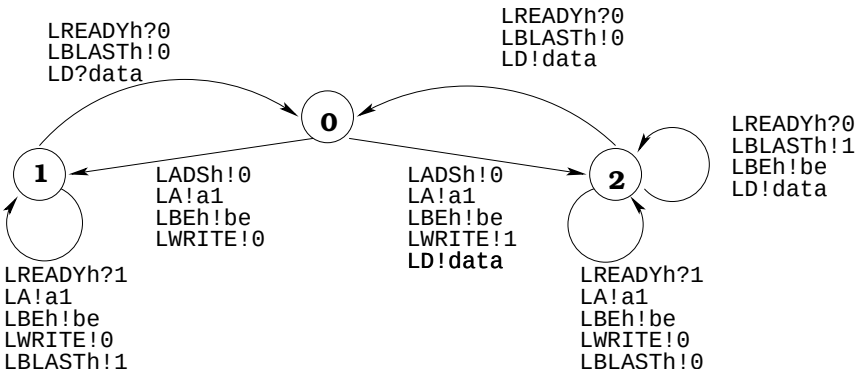
The algorithms developed have been implemented in a tool which was used to synthesise interfaces for common bus protocols. In particular, we have considered the AMBA Advanced High-performance Bus, AMBA Advanced System Bus, IBM CoreConnect Processor Local Bus and the OCPIP Open Core Protocol. To illustrate versatility of the algorithm, we have considered the protocols with different features as shown in Table 1. The protocols modelled have a maximum of 10 control states each. We have focused on read behaviours and have assumed that there are no zero latency transfers involved. The results of interface synthesis are presented in Table 2.

Master	Slave	Clock Ratios	Bus Width Ratios	I States	I Transitions
AHB	PLB	1:1	1:1	7	12
OCP	AHB	1:1	1:1	8	18
OCP	ASB	1:1	1:1	11	24
OCP	PLB	1:1	1:1	8	12
AHB	PLB	1:2	1:1	12	15
OCP	AHB	1:1	1:2	21	74
OCP	ASB	1:1	2:1	16	65
OCP	PLB	1:1	3:1	17	32

Table 2: Interfaces Synthesised



(a) Processor Local Bus



(b) Local Bus

Figure 9: Processor Local Bus and Local Bus

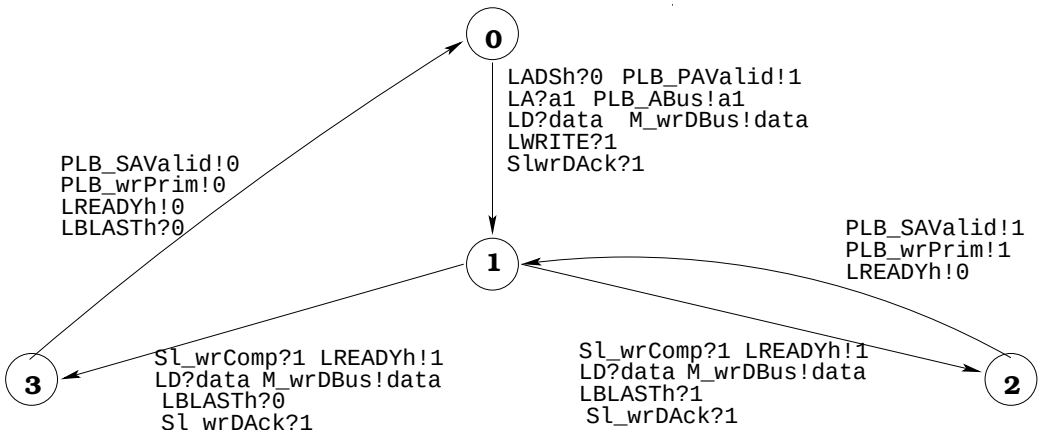


Figure 10: Interface for Processor Local Bus and Local Bus

4.1 Case Study

Along with the experimental results, we present a case study performed on a pair of protocols obtained directly from industry.

Figures 9(a) and 9(b) show state machines describing partial behaviour of the local bus from Alpha Data and the Processor Local Bus which forms a part of the CoreConnect bus architecture from IBM.

The Alpha Data Local Bus supports multiplexed address and data as well as pipelined operation, as shown in Figure 9(b) with all the signals involved in transactions. Signals which were suffixed by a # in the timing diagrams have been suffixed with h in the FSM. In states 1 and 2 there would also be transitions to state 0 labelled with the action `LBTERMh?0` indicating that the transaction may be terminated in which case a transition is made back to the initial state. These transitions are identical and hence have not been drawn to prevent cluttering. The part of the state machine containing only states 0 and 2 describe a write operation and the part containing states 0 and 1 describes a read operation. The two self loops in state 2 correspond to a wait cycle (`LREADYh?1`) and a pipelined read (`LBLASTh!0`). This FSM describes the protocol in detail.

In comparison, the Processor Local Bus(PLB) is a much more complex bus protocol but we consider only the small part of its functionality which is of interest. Figure 9(a) illustrates the PLB which allows a single cycle and pipelined writes. No other features are presented here as these features will not be required when interfacing with the Local Bus.

One possible interface between these two protocols is shown in Figure 10. The interface has four states. In constructing this interface, it has been assumed that the data and address buses of both protocols have equal width. Hence, the contents of the byte enable signal (`LBE`) have not been used in any transfers. The part of the interface involving transitions between states 0,1 and 3 allows for interoperability between non pipelined operation in both buses. The transitions between states 1 and 2 use the pipelined feature of the PLB only when the Local Bus requests a pipelined transfer.

The interface describes a method of ensuring correct transfers from one protocol to the other. There are many possible optimisations or implementations which can be made. For

example, it can be inferred from the interface state machine that the signal `PLB_PAValid` is a negation of `LADSh`. The signal `LBLASTh` can be latched and the output of the latch connected to `PLB_wrPrim` as the value of this signal in a cycle corresponds to that of `LBLASTh` in the previous cycle.

This FSM can be used to generate an RTL of the interface. In generating the RTL, the FSM component of the HDL code can be obtained by transcribing Figure 10. As the PLB has many signals, the designer should ensure that their values are maintained such that the described transfers can take place.

5 Conclusion

In this report, we have presented a general framework for modelling a range of hardware protocols and for addressing the problem of protocol mismatch. Specifically, the framework allows for detailed modelling of data buses, generation of provably correct interfaces and a novel technique for checking whether a pair of protocols match. The experimental results demonstrate that our framework is both easily adaptable to existing specification techniques and highly applicable to practical instances of the protocol mismatch problem. We extend the existing work in this area of research by addressing the issues of data type and bus width mismatches, transactions with multiple transfers, pipelined operation, complex branching structure in the protocols, differing clock speeds and provide solutions for the same.

Acknowledgements

This research was partially supported by a UNSW URSP grant (2002) and the Nicta Formal Methods group located at UNSW, where the third author was Visiting Professor from May to August 2003. We would like to thank Sri Parameswaran and his group for discussions and feedback and Partha Roop for alerting us to the interface synthesis problem.

References

[1] Virtual Socket Interface Alliance. http://www.vsi.org.

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

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

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

[5] G. Borriello, L. Lavagno, and R. 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] K. L. Calvert and S. S. Lam. Formal methods for protocol conversion. *IEEE Journal on Selected Areas in Communications*, January 1990.

[7] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer International Series in Engineering and Computer Science, 1994.

[8] IBM. 32-bit processor local bus, architecture specifications. *http://www-3.ibm.com/chips/products/coreconnect/*, Version 2.9.

[9] Open Core Protocol International Partnership Association Inc. http://www.ocpip.org.

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

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

[12] 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.

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

[14] A. Rajawat, M. Balakrishnan, and A. Kumar. Interface synthesis: Issues and approaches. *Proceedings of the 13th International Conference on VLSI Design*, January 2000.

[15] D. Shin and D. D. Gajski. Interface synthesis from protocol specification. *CECS Technical Report 02-13*, April 2002.

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

[17] Z. Tao, G. v. Bochmann, and R. Dssouli. A formal method for synthesizing optimized protocol converters and its application to mobile data networks. *Mobile Networks and Applications*, 1997.