Synchronous Protocol Automata: A Framework for Modelling and Verification of SoC Communication Architectures

Vijay D'silva
Indian Institute of Technology Bombay
vijay@cfdvs.iitb.ac.in

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

Arcot Sowmya
University of New South Wales, Sydney.
sowmya@cse.unsw.edu.au

Abstract

Plug-n-Play style Intellectual Property(IP) reuse in System on Chip(SoC) design is facilitated by the use of an on-chip bus architecture. We present a synchronous, Finite State Machine based framework for modelling communication aspects of such architectures. This formalism has been developed via interaction with designers and the industry and is intuitive and lightweight. We have developed cycle accurate methods to formally specify protocol compatibility and component composition and show how our model can be used for compatibility verification, interface synthesis and model checking with automated specification. We demonstrate the utility of our framework by modelling the AMBA bus architecture including details such as pipelined operation, burst and split transfers, the AHB-APB bridge and arbitration features.

1. Introduction

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[13] from IBM, AMBA[2] from ARM among others and the work of the VSI Alliance[1] and the OCP-IP[14] consortium. Figure 1 illustrates a typical bus-based SoC architecture. As IP blocks operate using different protocols and clock speeds, wrappers and bridges are introduced as shown to ensure inter-operability. Unfortunately, the vision of plug-n-play style assembly of SoCs is yet to be realised[3] as IP cores are designed with disparate protocols. IP integration involves *compatibility checking*

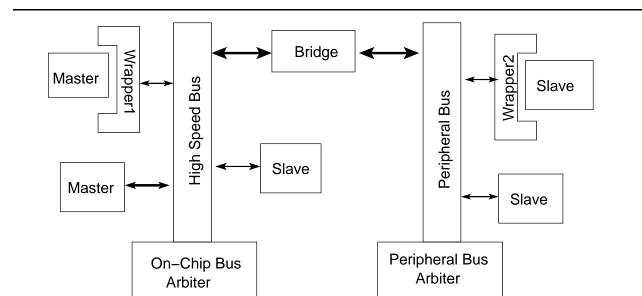


Figure 1. System-On-Chip Bus Architecture

between the IP protocols, *interface synthesis* to resolve protocol mismatches, component composition and system level verification. These additional steps increase the designers' effort and time required for chip design.

There have been research efforts towards modelling and formal verification of bus architectures[7, 11, 19], automated synthesis and verification of interfaces between mismatched protocols[6, 18]. These efforts have been independently motivated and the semantic disparity of the formalisms used impedes consolidation of solutions from the different areas. We present a formalism called *Synchronous Protocol Automata* which has been motivated by an extensive study of SoC bus protocols and interaction with designers and is capable of modelling complex protocol features while maintaining syntactic simplicity. This formalism has its foundation in the theory of synchronous languages[5, 16] and unifies seemingly independent existing work in integration and verification of bus based SoC architectures.

We have used this framework to model commonly used SoC bus protocols, to establish incompatibility and to synthesise interfaces for a protocol mismatch problem faced in the industry. Our experience has revealed that in contrast to timing diagrams, our model is more conducive to reasoning about different sequences of behaviours that protocols can exhibit. The modelling exercise also led to a con-

1530-1591/04 $20.00 (c) 2004 IEEE

cise and comprehensive documentation of the functional aspects of the protocols which can be used by designers.

1.1. Related Work and Overview

Different formalisms exist for modelling communicating hardware. The formalisms of Interface[8] and I/O automata[15] bear syntactic similarities to synchronous protocol automata but significant differences arise as we use a lower level of abstraction and synchronous semantics. While actions in these models represent methods and procedure calls, we use actions to describe the behaviour of hardware signals at a clock tick. Our automata consist of states which are either *input enabled* or *input insensitive*, an internal clock and control and data channels, all motivated by a hardware specific model.

The models used in the literature on interface synthesis [20, 17] are quite simplistic and informal and no criteria have been identified for interface correctness. In contrast, synchronous languages provide a precise semantic interpretation[4] for hardware behaviour and currently offer sophisticated tool support. Using synchronous protocol automata, we formalise various notions related to interface synthesis and indicate algorithms which apply within this framework.

There exists an independent body of exercises in formal verification of bus protocols including PCI[7], AMBA[19] and CoreConnect[11]. Such work involved using a model checking tool to verify that a model of the bus system satisfied a set of temporal logic specifications. Writing such specifications is a tedious task and the models used did not always capture all aspects of the protocols verified. As all existing formalisms suffer various deficiencies we propose a framework motivated by the requirements of the various problems studied.

1.2. Paper Contributions

In this paper, we present an automata-based framework for modelling all components of a bus architecture including bridges, wrappers, arbiters and components operating on multiple clocks. We provide a novel technique to establish compatibility of two SoC protocols, develop a method to reason about component composition and formalise the correctness criterion for interfaces used in IP integration. Using minimal input from the designer we can generate formal specifications and use them for system level verification. Synchronous protocol automata can easily be translated into any synchronous language or HDL enabling the designer to utilise the arsenal of tools available for these formats.

Following the suggestions of designers, we present a complete model and formal specification of the AMBA bus architecture which can be used by system builders and researchers. We are currently using this framework to generate concise documentations and formal specifications of commonly used protocols[2, 13, 14].

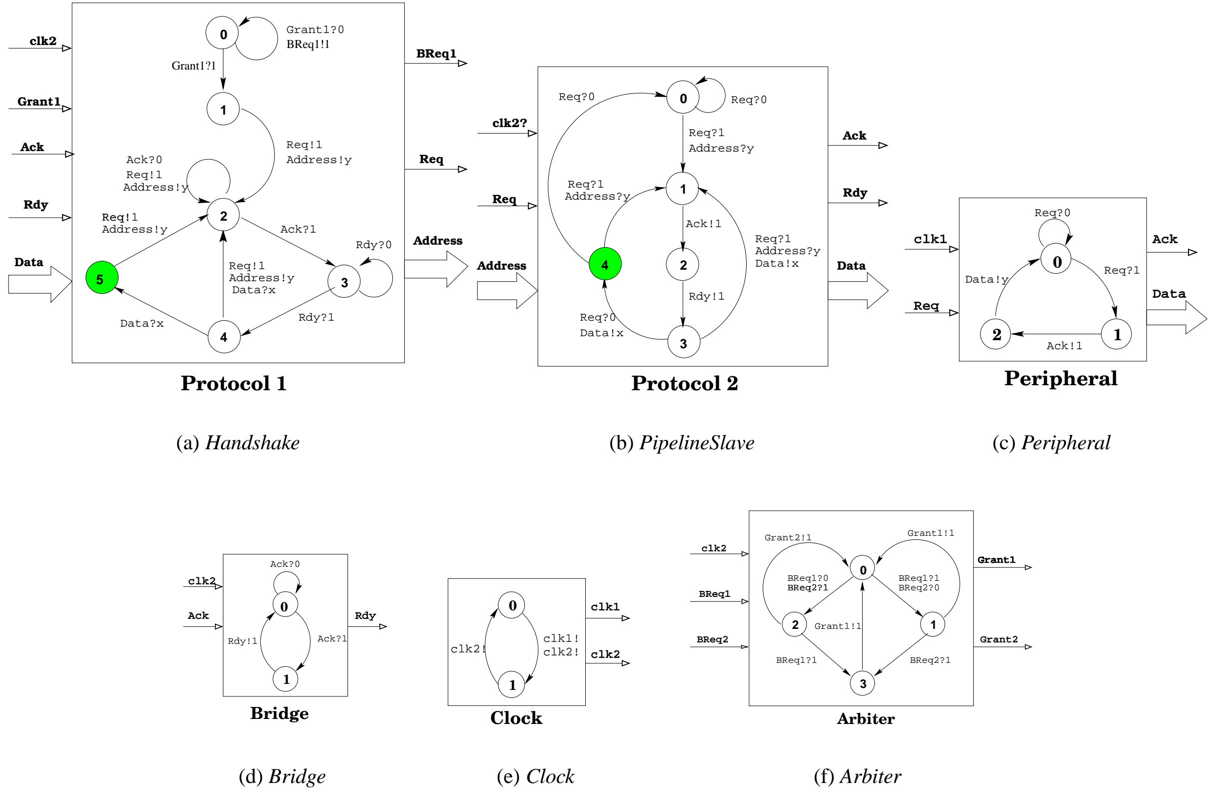
The paper is organised as follows: Section 2 presents an informal, complete overview of the proposed framework, Section 3 contains the formalisation of synchronous protocol automata, notions of composition, protocol mismatch and interface correctness and outlines the steps towards automated model checking. We illustrate the modelling and verification features with the AMBA bus protocol in Section 4 and conclude with discussions in Section 5.

2. Overview of Synchronous Protocol Automata

We illustrate this framework using a simple protocol called *Handshake* shown in Figure 3(a). *Handshake* receives input from the control channels `grant1`, `Ack` and `Rdy`, the data channel `Data` and can write to the control channels `BReq1`, `Req` and data channel `Address`. All transitions are synchronised with the protocol's clock $\mathtt{clk}_2$. *Handshake* makes a request for the bus and transits to state 1 when it is granted. It transits to state 2 at the next clock tick, raising a request (`Req!1`) and writing an address(`Address!y`) onto the bus. After an acknowledgement is received the protocol awaits an indication(`Rdy?1`) that data will be available. In the next cycle, the protocol may either read data and reach its *final state*, which is shaded, or pipeline the next request with the data read phase and transit directly to state 2. The choice between the outgoing transitions from state 4 is made by an internal condition which is not visible and makes the protocol appear non-deterministic. We call this kind of behaviour *weak determinism*. It can be observed that states 5 and 1 are equivalent or *bisimilar* when their outgoing transitions are compared. As the protocol would have had to perform at least one data operation to reach state 5, we may infer that one or more transfers has taken place. This feature of final states plays a key role in defining protocol compatibility even though a final state may be functionally redundant.

Figure 3(b) illustrates the protocol *PipelineSlave* which can pipeline its address and data phases as seen in the transition from state 3 to state 1. *Handshake* and *PipelineSlave* can communicate and progress in a lock step fashion. Once *Handshake* has been granted the bus, the states of the two protocols would go through the sequence $(1, 0)$, $(2, 1)$, $(3, 2)$, $(4, 3)$, $(5, 4)$. Since the two protocols are able to communicate once the bus is granted, we call them a *matched* protocol pair.

We now consider a protocol *Peripheral* in Figure 3(c) which operates at half the speed of *Handshake* with a clock $\mathtt{clk}_1$. The bridge in Figure 3(d) facilitates communication between these protocols and Figure 3(e) is the system clock. Such clock models can be constructed for systems with clocks derived from a single source having arbitrary ratios



(a) *Handshake* (b) *PipelineSlave* (c) *Peripheral*

(d) *Bridge* (e) *Clock* (f) *Arbiter*

to each other. Figure 3(f) is an example of a bus arbiter which accepts only one request in a given cycle.

These automata describe a system with two masters and a slave on a high speed bus, a slave on a low speed bus, an arbiter and a bridge. Once all components in the system have been modelled, compatibility checks can be enforced. Additional aspects of correctness can be checked by translating the automata into the input language of a model checker. Properties that the designer may wish to ensure include mutual exclusion in bus ownership specified as $AG\neg(\texttt{Grant}_1 \wedge \texttt{Grant}_2)$ and liveness which includes the specification $AG(\texttt{BusReq}_2 \rightarrow AF\texttt{Grant}_2)$. This liveness property will fail as the arbiter in this system may ignore $\texttt{Grant}_2$ forever. As such properties would be required to hold in all bus protocols, it is sufficient for the designer to identify the grant, request and acknowledge signals to automate property generation. More complex specifications are explored in sections 3 and 4.

3. Formal Definitions

Definition 1 *A Synchronous Protocol Automaton P is defined as a tuple $(Q, \Sigma, \Delta, V, \mathcal{A}, \longrightarrow, clk, q_0, q_f)$. 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 with one for each data channel and $\mathcal{A}$ is the set of actions which can be performed. $\longrightarrow \subseteq Q \times clk? \times \mathcal{A} \times Q$ is the state transition relation. q_0 and q_f are the initial and final states.*

Channels are typed and unidirectional. Control channels are usually boolean while the type of a data channel may vary. An action S in $\mathcal{A}$ is of the form $G_1 \cdot N_1 \ldots G_k \cdot N_k$ where each G_i is a set of guards or *blocking operations* and N_i is a set of *nonblocking operations* which can be performed. Guards check for the value of a signal(denoted $c?v$) or the absence of an event on a signal (high impedance), denoted $c\#$ on a control channel c. Nonblocking operations are writes on control channels, denoted as $c!v$ and reads or writes on data channels, denoted as $d!v, d?x$. τ is the empty action. Typical transitions in bus protocols would have only one guard and response as seen in Figure 2. A sequence of guards and responses occurs when signals are exchanged using combinational logic. Such behaviour is exhibited by *zero-latency* transfers as present in the Open Core Protocol[14].

A transition is written as $q \xrightarrow{S} q'$ and a sequence of transitions as $q \overset{\alpha}{\Longrightarrow} q'$ where $\alpha = S_1, S_2 \ldots S_n$. A *transaction run* is a transition sequence $q_0 \overset{\alpha}{\Longrightarrow} q_f$ in which a transfer is initiated and completed. Any path from state 0 to state 5 in *Handshake* is a transition run.

The predicate $\texttt{blocking}(q)$ is true in a state q if all outgoing transitions are guarded. q is non-blocking if all outgoing transitions are unguarded. When the automaton is in a state, the transition whose guard evaluates to true is taken at the clock tick. If more than one guard is true, a nondeterministic choice is made. We require that all states be ei-

ther blocking or nonblocking. Our focus is restricted to a subclass of nondeterministic automata which are *weakly deterministic* like *Handshake*. A protocol is weakly deterministic if transitions to multiple target states are distinguishable by their output or lead via identical transitions to states distinguishable in this manner. A state with two different data transitions is non-deterministic but not weakly deterministic. These restrictions reflect the general structure of bus protocols and are unique to our formalism.

Definition 2 *Two non-blocking states q_1, q_2 are* output distinguishable *if for any $q_1', q_2', S_1, S_2 : q_1 \xrightarrow{S_1} q_1', q_2 \xrightarrow{S_2} q_2'$, there exists at least one control channel c such that $c! \in S_1$ and $c! \notin S_2$ or vice versa.*

Definition 3 *A protocol is weakly deterministic if whenever $q_0 \stackrel{\alpha}{\Longrightarrow} q_1 \xrightarrow{S_1} q_1'$ and $q_0 \stackrel{\alpha}{\Longrightarrow} q_2 \xrightarrow{S_2} q_2'$ and $q_1 \neq q_2$ and $S_1 \neq S_2$ then, q_1 and q_2 are output distinguishable.*

3.1. Protocol Compatibility

Given models of the communicating components in a SoC architecture, it is of great importance to ensure that data is always transferred as required and that deadlocks do not occur. Intuitively, at any clock tick, the actions that a pair of protocols attempt to perform should permit both of them to progress. We use the predicate `permit` and a *transaction relation* defined below to formalise these notions.

Definition 4 *A causal dependency graph between a pair of actions S_1 and S_2 is constructed by adding a directed edge from G_i to N_i if $G_i{\cdot}N_i \in S_1$ and from N_i to every $G_j' \in S_2$ such that $c! \in N_i$ and $c? \in G_i'$ where c is a control channel.*

Definition 5 $\texttt{permit}(S_1, S_2)$ *holds for a pair of actions if their causal dependency graph is acyclic and for every $c?, c'\# \in S_1$, $c! \in S_2, c'! \notin S_2$ and vice versa where c is a control or data channel.*

Intuitively, $\texttt{permit}(S_1, S_2)$ holds for a given pair of actions if for every read operation in one action, a write exists in the other and vice versa. Causality cycles are eliminated by using the causal graph. Using $\texttt{permit}(S_1, S_2)$ we may establish that two actions are *compatible* with each other. In order to establish compatibility between two automata P_1 and P_2 with final states r_f and t_f respectively, we define a relation on their states.

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 $r \xrightarrow{S_1} r'$ and $t \xrightarrow{S_2} t'$, $\texttt{permit}(S_1, S_2)$ holds 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 $r \xrightarrow{S_1} r'$ there exists $S_2, t' : (t \xrightarrow{S_2} t'$ and $\texttt{permit}(S_1, S_2))$ and for all such $S_2, t' : \langle r', t' \rangle \in R$*
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 $\texttt{permit}(S_1, S_2)$, $\langle r', t' \rangle \in R$*

The first condition matches the final states of two protocols. The second condition ensures that if both protocols perform only data operations they operate on the same channels and the third ensures that each guard in a transition in one protocol is satisfied by some action of the other. The last condition states that if both protocols have a default guard which is true they should transit simultaneously to matched states. This situation would be extremely rare in a real protocol but has been added for completeness.

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 $\langle r_0, t_0 \rangle \in \mathcal{R}$.*

If such a relation does not exist, the protocol pair is mismatched and an interface has to be synthesised. An interface can also be modelled as a synchronous protocol automaton I. I can be composed with either protocol, say P_2 to get a new automaton denoted $I||P_2$ where $||$ is a synchronous parallel composition operator. We formalise the notion of interface correctness which applies to wrappers and bridges as well.

Definition 8 *An interface I between two mismatched protocols P_1 and P_2 is* correct *if there exists a transaction relation $\mathcal{R}$ between the initial states of P_1 and $I||P_2$.*

Correctness can be defined equivalently in terms of $P_1||I$ and P_2. Interfaces can be synthesised using techniques presented in [6, 17, 20]. We present a synthesis technique which is correct by construction in [10]. Synchronous parallel composition is defined below.

Definition 9 *The synchronous parallel composition $P_1||P_2$ of two synchronous protocol automata P_1 and P_2 with a shared set of channels $C = (\Sigma_1 \cap \Sigma_2) \cup (\Delta_1 \cap \Delta_2)$ is an automaton defined as*

- $Q_{P_1||P_2} = Q_1 \times Q_2$
- $\Sigma^I_{P_1||P_2} = (\Sigma^I_1 \cup \Sigma^I_2) \backslash C$, $\Sigma^O_{P_1||P_2} = (\Sigma^O_1 \cup \Sigma^O_2) \backslash C$
- $\Delta_{P_1||P_2} = (\Delta_1 \cup \Delta_2) \backslash C$, $\mathcal{A}_{P_1||P_2} = (\mathcal{A}_1 \cup \mathcal{A}_2) \backslash \mathcal{A}_C$
- *$\langle r_0, t_0 \rangle$ is the initial state and $\langle r_f, t_f \rangle$ is the final state*
- *$\longrightarrow_{P_1||P_2}$: given $r \xrightarrow{S_1} r'$ and $t \xrightarrow{S_2} t'$ and actions S_1', S_2' which are the projections of S_1, S_2 on $\mathcal{A}_C$ the following rule defines $\longrightarrow_{P_1||P_2}$*

$$\frac{r \xrightarrow{S_1} r' \wedge t \xrightarrow{S_2} t' \wedge \texttt{permit}(S_1', S_2')}{\langle r, t \rangle \xrightarrow{S_c} \langle r', t' \rangle}$$

where $S_1 = G_1{\cdot}N_1.S_3$ and $S_2 = G_2{\cdot}N_2.S_4$ and $S_c = G_1 \cup G_2 \backslash \mathcal{A}_C {\cdot} N_1 \cup N_2 \backslash \mathcal{A}_C.S_c'$ and S_c' is defined in a similar manner.

The communication infrastructure of the SoCs considered are completely synchronous and can be represented as $\mathcal{C}lk||P_1||\ldots||P_k$ where $\mathcal{C}lk$ models the system clock and P_i may be a master, slave, arbiter, slave, wrapper or bridge. A more elaborate treatment of the formalism presented with illustrative examples is available in[9].

3.2. Model Checking with Automated Property Extraction

A set of synchronous protocol automata describing a bus architecture can be translated into a set of concurrent communicating processes described in languages used by a model checker. As this framework is being applied to bus protocols, we have identified properties which are routinely verified and can generate them automatically based on input from the designer and thereby alleviate the specification effort required. In addition to properties of the kind shown in Section 2, a few other templates can be written as well. A,G and F are well known temporal logic operators.

1. $AG(\texttt{Grant}_i \leftrightarrow AF\texttt{BusReq}_i)$: Every request for the bus should eventually be granted and vice versa. This requirement is quite strong and may not always be satisfied. The AMBA AHB protocol allows for a master to be granted the bus without a preceding request.
2. $\texttt{Wait}\,\mathcal{U}(\neg\texttt{Wait} \vee \texttt{Abort})$: All wait cycle sequences should be finite and should be terminated either normally or by a transfer abort.

The designer will be required to identify signals in the system based on their functionality and examples include the signals for `BusRequest, BusGrant, DataAcknowledge, BurstTransfer` and `Wait`. These signals can be cast into templates to generate temporal logic specifications. The specification and system model can then be used for verification.

The methodology developed so far illustrates how this framework can be used for modelling, automated verification and to formalise notions of composition, compatibility, and various notions of correctness.

4. The AMBA bus protocol

We now present a case study in modelling and verification with the Advanced Microcontroller Bus Architecture(AMBA) from ARM which is widely used in the industry.

The model was constructed through a study of the AMBA specification document and discussions with designers who had implemented it. Figure 3 is a fragment of the AMBA High Speed Bus(AHB) protocol. The sub-machine between states $2, 3, 4$ and 5 captures transfers with incremental bursts of undefined length. The protocol fragment also captures transfers of type `NON-SEQ`(nonsequential), `SEQ`(sequential) and `BUSY`. States 3 and 5 are identical but have been drawn separately for presentation purposes. The transitions to state 6 indicate the beginning of a 4 beat wrapping burst. The shaded triangle indicates that the entire automaton has not been drawn. While there are many transitions corresponding to the different types of transfers and responses which are possible, the general structure of the entire protocol resembles that shown in Figure 3. The state machine had 18 states and 40 transitions. The automaton describing the AHB read transfers is smaller, as most transitions are contingent on the behaviour of the slave which is less varied than that of the master as it has fewer signals.

In contrast, AMBA Peripheral Bus(APB) shown in Figure 4 is extremely small with only three states as it has only

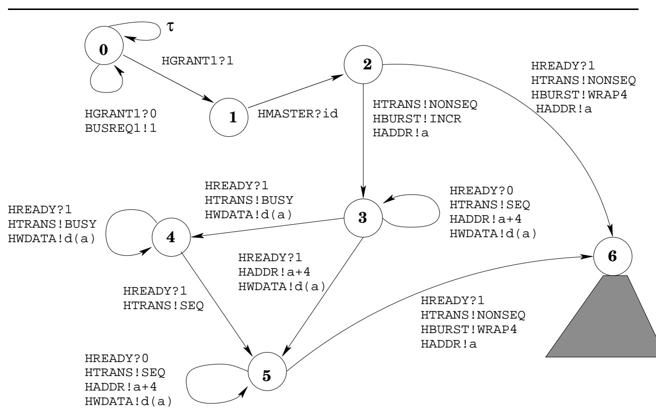


Figure 3. AMBA Master for write transfers

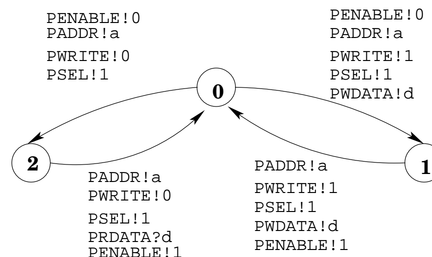


Figure 4. APB Master

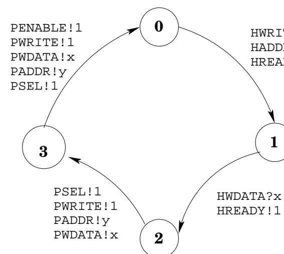


Figure 5. AHB to APB bridge

two control signals and had no pipelined operation. Figure 5 models the bridge between the AHB and APB buses for write transfers.

Using this model and requirements stated in the AMBA specification[2], a formal specification of the protocol was created. Some interesting properties are discussed below. The page numbers refer to the AMBA specification.

- $AG(\texttt{Increment} \rightarrow \neg(\texttt{Increment}\ \mathcal{U}\ \texttt{Boundary}))$: Captures the requirement on page 46 that *An incrementing burst can be of any length but .. must not cross a 1kB boundary*.
- $AG(\texttt{SlaveRead} \rightarrow HREADY = 1)$: From page 53 that *A slave must only sample the address and control signals and* HSELx *when* HREADY *is high*.
- Certain specifications like : *it is recommended,..,that slaves do not insert more than 16 wait states* led to cumbersome formulae. It was easier to write protocol automata describing these properties. A single protocol automaton can be written to capture all properties pertaining to burst constraints. This automaton can then be used to *observe* if the architecture satisfies a given property by composing it with the system and checking to see if *bad states* are reachable. This style of *observer based verification* is well known in the synchronous language literature[12].

5. Conclusions

In this paper we used Synchronous Protocol Automata to formalise the notions of protocol mismatch, interface correctness and component composition and showed how these methods lend themselves to interface synthesis and model checking. This formalism has been used to document and formally specify the AMBA protocol. Complete documentation and specifications of other bus architectures such as the CoreConnect and the Open Core Protocol will be available soon. We have applied our framework successfully in an industrial setting. This formalism has proved to have a high utility value and we are currently focusing on providing tool support for the methods described here.

Acknowledgements

This work was partially supported by UNSW USRP Grant 2003, NICTA, Sydney and CFDVS IIT Bombay. We would like to thank Sri Parameswaran and his group for their feedback and discussions.

References

[1] V. S. I. Alliance. http://www.vsi.org.

[2] ARM. Advanced micro-controller bus architecture specification. *http://www.arm.com/armtech/AMBA_spec*, 1999.

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

[4] G. Berry. A hardware implementation of pure esterel. *Sadhana, Academy Proc. in Engineering Sciences*, 1992.

[5] G. Berry. The foundations of esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.

[6] G. Borriello, L. Lavagno, and R. B. Ortega. Interface synthesis: a vertical slice from digital logic to software components. *Int'l Conf. of Computer-Aided Design*, November 1998.

[7] P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying ip-core based system-on-chip design. *IEEE ASIC.*, September 1999.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. *Joint 8th ESEC and 9th ACM Symposium on FSE*, 2001.

[9] V. D'silva, S. Ramesh, and A. Sowmya. Automated interface synthesis. *School of Computer Sci. and Eng., Technical Report 0325*, University of New South Wales 2003.

[10] V. D'silva, S. Ramesh, and A. Sowmya. Bridge over troubled wrappers:automated interface synthesis. *17th IEEE Int'l Conference of VLSI Design*, January 2004.

[11] A. Goel and W. R. Lee. Formal verification of an ibm coreconnect processor local bus arbiter core. *37th Design Automation Conference*, June 2000.

[12] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. *Algebraic Methodology and Software Technology*, 1993.

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

[14] O. C. P. I. P. A. Inc. Open core protocol specification. *http://www.ocpip.org*, Release 1.0, 2001.

[15] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *6th ACM Symposium on Principles of Distributed Computing*, 1987.

[16] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. *Int'l Conference on Concurrency Theory*, August 1992.

[17] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. *Int'l Conf. on Computer-Aided Design*, November 2002.

[18] A. Rajawat, M. Balakrishnan, and A. Kumar. Interface synthesis: Issues and approaches. *13th Int'l Conf. on VLSI Design*, January 2000.

[19] A. Roychoudhury, T. Mitra, and S. R. Karri. Using formal techniques to debug the amba system-on-chip bus protocol. *Design Automation and Test in Europe*, March 2003.

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