

Verification of AMBA Using a Combination of Model Checking and Theorem Proving

Hasan Amjad¹

*Computer Laboratory
University of Cambridge*

Abstract

The Advanced Microcontroller Bus Architecture (AMBA) is an open System-on-Chip bus protocol for high-performance buses on low-power devices. We demonstrate the combined use of model checking and theorem proving to verify both control and datapath properties in a seamless manner.

Key words: System-on-Chip, theorem proving, model checking, tool combination.

1 Introduction

Typical microprocessor and memory verifications assume direct connections between processors, peripherals and memory, and zero latency data transfers. They abstract away the data transfer infrastructure as it is not relevant to the verification. However, this infrastructure is in itself quite complex and worthy of formal verification.

The Advanced Microcontroller Bus Architecture² (AMBA) [2] is an open System-on-Chip bus protocol for high-performance buses on low-power devices. In this paper we implement a simple model of AMBA and verify latency, arbitration, coherence and deadlock freedom properties of the implementation.

The verification is conducted using HOLCHECK, a model checker for the propositional μ -calculus L_μ [8], that is part of the HOL theorem prover [1]. This allows results from the model checker to be represented as HOL theorems for full compositionality with more abstract theorems proved in HOL using a formal model theory of L_μ that we have also developed. This tight connection between model checking and theorem proving is exploited in sections 4 and 5 of this work.

¹ Email:ha227@c1.cam.ac.uk

² ©1999 ARM Limited. All rights reserved. AMBA is a trademark of ARM Limited.

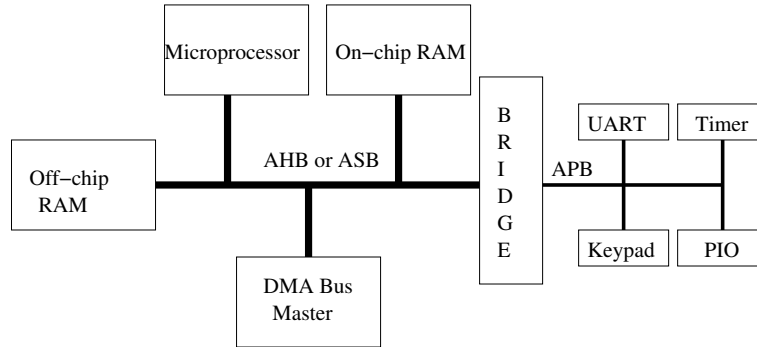


Fig. 1. Typical AMBA-based Microcontroller

2 AMBA Overview

The AMBA specification defines the following buses:

- **Advanced High-performance Bus (AHB):** The AHB³ is a system bus used for communication between high clock frequency system modules such as processors and on-chip and off-chip memories. The AHB consists of bus masters, slaves, an arbiter, a signal multiplexer and an address decoder. Typical bus masters are processors and DMA devices.
- **Advanced Peripheral Bus (APB):** The APB is a peripheral bus specialised for communication with low-bandwidth low-power devices. It has a simpler interface and lower power requirements.

The APB has a single bus master module that acts as a bridge between the AHB and the APB. The AMBA specification is hardware and operating system independent and requires very little infrastructure to implement. Figure 1 shows a typical AMBA-based microcontroller. We follow revision 2.0 of the AMBA specification [2].

3 Specification

The AMBA AHB and APB specification is a 110 page document. Due to space constraints, we can only give a brief summary here.

3.1 AMBA APB

The APB is used for connecting the high-bandwidth AHB system bus to low-bandwidth peripherals such as input devices. There is a single bus master, a single global clock, and all transfers take two cycles. The bus master also acts as a bridge to the system bus, to which it can be connected as a slave. The address and data buses can be up to 32 bits wide.

³ The AHB supersedes the AMBA Advanced System Bus (ASB) and is superseded by the AMBA Advanced eXtensible Interface (AXI).

The operation of the APB consists of three stages, all of them are triggered on the rising edge of the clock:

- (i) *IDLE*. This is the initial and the default state of the bus when no transfer is underway i.e. all slave select signals are low.
- (ii) *SETUP*. The first stage of a transfer is a move to the SETUP state, signalled by a slave select signal going high. The address and data signals are asserted during this phase. The direction of transfer (read/write) is indicated by another signal. This stage always lasts for one clock cycle and then the operation moves to the ENABLE stage.
- (iii) *ENABLE*. The address, data and control signals are stable during this phase. This phase also lasts one clock cycle and then moves to the SETUP or the IDLE stage.

3.2 AMBA AHB

The AHB is a pipelined system backbone bus, designed for high-performance operation. It can support up to 16 bus masters and slaves that can delay or retry on transfers. It consists of masters, slaves, an arbiter and an address decoder. It supports burst and split transfers. The address bus can be up to 32 bits wide, and the data buses can be up to 128 bits wide. As before, there is a single global clock. We have not described several implementation details due to space considerations.

The operation of the AHB is too complex to be specified in terms of a few fixed stages. A simple transfer might proceed as follows (the list numbering below is not cycle-accurate):

- (i) The AHB is in the default or initial state. No transfer is taking place, all slaves are ready and no master requires a transfer.
- (ii) Several masters request the bus for a transfer.
- (iii) The arbiter grants the bus according to some priority-scheduling algorithm.
- (iv) The granted master puts the address and control information on the bus.
- (v) The decoder does a combinatorial decode of the address and the selected slave samples the address.
- (vi) The master or the slave put the data on the bus and it is sampled. The transfer completes.

Items 4-5 above constitutes the *address phase* of a transfer, and 6 constitutes the data phase. Since the address and data buses are separate, the address and control information for a transfer are driven during the data phase of the previous transfer. This is how transfers are pipelined. Several events can complicate the basic scenario above e.g. extended transfers (slave inserts wait states), bursts (multiple transfers without renegotiating for bus ownership), splits (master put on hold by slave), retries (master asked to try later)

and aborts (slave signals failure).

Masters

The AHB supports up to 16 bus masters. Each master wishing to initiate a transfer competes for a bus grant from the arbiter and has its control and address signals driven to the slave when it gets the bus.

If a master x does not wish to initiate a transfer it drives its bus request signal to low and if it owns the bus it also signals IDLE. To initiate a transfer, it drives its bus request signal to high. Upon getting bus ownership (the arbiter decides this), the address and control signals are driven onto the bus for exactly one cycle. To initiate the transfer, the master signals NSQ (which abbreviates “non-sequential”). It also drives the burst signal to low indicating a single transfer, or to high indicating a multi-beat burst. All this happens during the one-cycle address phase.

In the next cycle, the master drives the data on to the data buses (or samples it in case of a read). If this is a burst, then the master also continues to drive the control signals and to increment the address signals to prepare for the next beat of the burst. From the second beat onwards, the master signals SEQ (“sequential”) rather than NSQ.

Masters also need to respond to slave signals for extended transfers (usually by signalling IDLE), splits (by suspending the current transfer until the slave is ready), retries (aborting and requesting the bus again) and aborts (aborting the transfer).

Multiplexer

The bus uses a central multiplexer interconnect scheme. All masters drive their address and control signals and the arbiter decides which master’s signals are routed on to the slaves.

Arbiter

The arbiter uses an arbitration algorithm (e.g. round-robin scheduling; AMBA does not specify or recommend any particular algorithm) to decide which master to grant the bus to. Actual bus ownership is not handed over until the current transfer completes. Additionally, the arbiter is responsible for keeping track of masters (by internally masking their bus requests) that have split transfers outstanding and granting the bus to the highest priority one when the corresponding slave signals that it is ready to continue the transfer.

Decoder

The decoder simply performs a direct decode of the address bus. The appropriate higher order bits indicate the selected slave and the rest are used by slaves to determine source/target registers.

Slaves

Once a transfer begins it is up to the slave to determine how it proceeds. The slave can do one of the following:

- If all is well, the slave responds by signalling READY and OK, and the transfer is straightforward.
- If the slave needs a little time during the data phase, it can extend the phase by inserting wait states by driving READY to low and signalling OK. Note that the address phase cannot be extended.
- If the slave cannot complete the transfer immediately it can issue a SPLIT response, if it is SPLIT-capable.
- If a non-SPLIT-capable slave cannot complete a transfer immediately it signals RETRY. Retried transfers must be restarted from scratch.
- In case of a complete failure, the slave signals to ERROR, and ignores the rest of the transfer.

The RETRY, SPLIT and ERROR responses take two cycles (READY is low in the first cycle, high in the second), to give the master time to re-drive the address and control signals onto the bus.

4 Implementation

We implement the APB by following the specification in a straightforward manner without any optimizations.

4.1 Assumptions

We make some simplifying assumptions:

- (i) All signals are valid throughout, i.e. there is no glitching. Sub-cycle timing (i.e. timing delays between signals becoming stable after changing) is ignored.
- (ii) Since there is a single global clock triggering all signals, transitions of the state machine are synchronous. For the same reason, it suffices to model the clock implicitly by equating one transition of the system to one clock cycle.
- (iii) Endian-ness is not fixed, but is assumed to be consistent throughout.

These assumptions preserve the properties of the model that we are interested in. This is because all properties are at the signal level, and because the specification itself uses a single clock and does not specify endian-ness.

For AHB, we restrict the model further:

- (i) We model 8 masters. The specification places an upper bound of 16 on the number of masters.
- (ii) We model 16 slaves. The specification places no upper bound on the

number of slaves.

- (iii) Slaves may split on at most one master. The specification recommends (but does not require) that all split-capable slaves be able to split on the maximum number of masters.

The bound in the second assumption is arbitrary and increasing it appears not to significantly tax our model checker. The third assumption is due to constrained development time. The first assumption is the only one forced upon us: increasing the number of masters causes an exponential increase in the size of the state space and our system cannot cope. This is likely because HOLCHECK currently lacks standard model checking optimizations such as partitioning and symmetry reductions. We note however that earlier AMBA verifications used considerably more constrained models [11,12].

4.2 The Formal Model

We convert the specification to a formal model in HOL, expressed as a set of definitions. The internal behaviour of the master and slaves has been abstracted away, as being irrelevant to the verification. Since this is a hardware model, and since it will eventually be model checked, it is convenient to model the system as a state machine.

We define our state machines for APB and AHB, M_{APB} and M_{AHB} respectively. Signals are modelled as a boolean variables, and a state $\bar{s}_{APB/AHB}$ is represented as a tuple of these variables. We define an initial states predicate $S_{0APB/AHB}$ on states, and a transition predicate $R_{APB/AHB}$ on states and “next” states. As the model is synchronous, $S_{0APB/AHB}$ and $R_{APB/AHB}$ are just the conjunctions of the respective predicates for the system modules, such as masters, slaves and the arbiter. Our system internally converts these predicates to a formal Kripke model, preparatory to model checking.

The full formal model is about 1600 lines of HOL definitions. Due to space constraints, we illustrate the formal modelling using the definitions for the AHB arbiter (we implement a simple priority-based one), and present notable features from other definitions. The full model will be made available with the upcoming Kananaskis-3 release of HOL.

Definition 4.1 $S_{0AHB}^{arb}(\bar{s}_{AHB})n = GRANT\ n \wedge MASTER\ n$

The initial states predicate for the arbiter is parameterized by the total number of masters less one. It says that we start with the highest priority master being granted ($GRANT\ n$) and having bus ownership ($MASTER\ n$). There are 8 masters, numbered from 0 to 7 (in increasing order of priority). Thus we instantiate this predicate with $n = 7$. We take some liberty with the formal notation here: $GRANT\ n$ in HOL is actually $GRANT\ n\ (grant_0, \dots)$ where the tuple contains all the boolean variables $grant_i$ corresponding to arbiter grant signals, and similarly for the other signals.

The transition relation is more complex:

Definition 4.2

$$\begin{aligned}
R_{AHB}^{arb}(\bar{s}_{AHB}, \bar{s}'_{AHB})n = & \\
& (\text{if } n = 0 \text{ then } GRANT' 0 \\
& \text{else if } MASK\ n \wedge BUSREQ\ n \wedge IDLE \text{ then } GRANT' n \\
& \quad \text{else } R_{AHB}^{arb}(\bar{s}_{AHB}, \bar{s}'_{AHB})(n - 1)) \wedge \quad (1) \\
& (MASTER' n = \text{if } READY \text{ then } GRANT\ n \text{ else } MASTER\ n) \wedge \quad (2) \\
& (\text{if } SPLIT \wedge \neg READY \wedge MASTER\ n \text{ then } MASK' n \\
& \text{else if } HSPLIT\ n \text{ then } \neg MASK' n \\
& \quad \text{else } MASK' n = MASK\ n) \quad (3)
\end{aligned}$$

This predicate is also parameterized by the total number of masters less one. We prime a predicate to denote that its elided second argument (the boolean tuple) consists of primed next-state variables. Conjunct (1) says that the highest priority master that requests the bus is to be granted the bus, provided the bus is idle and the master is not masked. Conjunct (2) says the currently granted master is to be given bus ownership when the last active slave has signalled READY (indicating completion of transfer). Conjunct (3) says that a split master should be masked, or else unmasked if the slave that signalled the split signals the end of the split via HSPLIT.

The use of the theorem prover’s higher-order logic as our modelling language allows us to define components in a natural manner, using recursion, parameterization and higher order predicates. A straightforward invocation of the HOL simplifier unfolds an instantiated version of this definition into completely boolean model-checkable form, by expanding out the recursion and rewriting out the definitions of the predicates.

Another instance of this automation is the following conjunct from the definition of a split-capable slave

$$\forall m. m < 8 \Rightarrow \neg HSLVSPLIT\ n\ m \Rightarrow \neg HSPLIT' m$$

which prohibits a slave n from asserting the “unsplit now” HSPLIT signal for a master m , if n was not the slave that split on m originally ($HSLVSPLIT\ n\ m$). This line showcases the use of bounded quantification as well as of predicates with multiple parameters.

In the definition of the arbiter, we used MASK signals, which are not explicitly mentioned in the specification as part of the arbiter control interface. However, their use is required for a correct implementation. Since we are using HOL, abstracting the MASK signals out of the arbiter interface is simply a matter of existentially quantifying the arbiter definition with respect to these signals. So we have a straightforward method for hiding details that are necessary for a correct model, but are of no concern to clients of the module.

Being able to use hidden variables, recursion, parameterization, abbreviations and bounded quantification as modelling constructs is not a novelty of our system. However, we do get readability, automation and a proof of cor-

rectness of the automatic translation between the human-readable (and hence human prover usable) and the model-checkable forms of the model, simply by virtue of integration with HOL, with no extra development effort. This is one of the many ways in which using a tight integration of model checking with a theorem prover pays off.

5 Verification

CTL [3] is a temporal logic commonly used in symbolic model checking. We model check various standard bus architecture CTL properties for our implementation, by translating them to L_μ via HOL. As usual, a property is considered verified if the set of satisfying states include the initial states.

5.1 Datapath compression

Model checkers typically run into trouble when the datapath is introduced, due to the sheer number of new state variables. However, we observe that in the APB (and indeed the AHB) implementation, data and address signals are only copied around and their actual values do not influence system behaviour (with the exception of some of the higher bits of the address used by the decoder, but these can be split away or the decoding abstracted), i.e., they are *data independent*. This suggests a simple abstraction that significantly reduces the number of state variables.

Our data and address signals only ever occur in equality tests (or can be formally rewritten to be so). If there are n such tests, then allowing each signal to range over a finite type with n values is sufficient to allow all possible combinations of successes and failures of the equality tests. If a signal can only have n values, it can be modelled in the standard way by $\text{ceil}(\log_2 n)$ boolean variables.

For example, our APB implementation has six address and data signals, 12 including the next-state versions. Thus any possible combination of equality tests over these signals requires them range over at most 12 values. Hence each signal can be modelled with four boolean variables, rather than 32 or 64.

We are in the process of formalising this reasoning in HOL. This belongs to a well-known class of model checking abstractions known as symbolic abstractions [6]. Our novelty lies in the formalization and automation aspects.

5.2 Verifying APB

5.2.1 Latency and Coherence

Latency properties check that the bus becomes available within a given number of cycles. We can use them to check that wait and/or transfer times do not exceed design specifications. Coherence properties check data coherency, i.e. registers are updated correctly at the end of transfers. Since transfers are multi-cycle, target registers are not updated immediately. Thus by checking

that the update happens in precisely two cycles, we can also check the transfer time. The embedded model checker returns the required result as a formal HOL theorem:

Theorem 5.1

$$\vdash \forall \bar{s}_{APB}.$$

$$\bar{s}_{APB} \models_{M_{APB}}$$

AG

$$\begin{aligned} & ((\neg PENABLE \wedge PSEL_x \wedge PWRITE) \\ & \quad \Rightarrow ((\mathbf{AXAX}SDATA_x) = MDATA)) \\ \wedge & ((\neg PENABLE \wedge PSEL_x \wedge \neg PWRITE) \\ & \quad \Rightarrow (SDATA_x = (\mathbf{AXAX}MDATA))) \end{aligned}$$

Here $PSEL_x$ denotes that slave x is selected, $PENABLE$ high indicates that we have reached the third stage of the APB transfer, $PWRITE$ indicates whether the transfer is read or write, and $SDATA_x$ and $MDATA$ represent the source/target registers for the slave and master respectively.

Note that the theorem statement as written is ill-typed with respect to CTL semantics. In particular, $\mathbf{AXAX}SDATA_x$ is ill-typed, since $SDATA_x$ is not a proposition. This is notational abuse, abbreviating the fact that the property was actually checked for each bit representing the datapath variables. Thanks to datapath compression, the model checker was able to cope.

Henceforth we shall simply state the property that was verified, as all formal theorem statements follow the same pattern.

5.2.2 Deadlock Freedom

In concurrency theory, the term *deadlock* refers to an abnormal termination or freeze of the system. In terms of automata such as Kripke structures, this may be represented by a state with no outgoing transitions.

We can check that this undesirable situation does not occur. Our APB transition relation has been defined by assigning all next-state variables some value in each cycle, so the simple CTL property

$$\mathbf{AG EX True}$$

(to check that there is no terminal state) is in a sense vacuously true and does not tell us anything.

On account of this, we need to have some criterion for system deadlock. We know that once a transfer is underway, it always completes, by Theorem 5.1. So it remains only to check that a transfer can always be initiated. This can be checked by the following property schema:

$$\mathbf{AG}(\mathbf{AF}(PSEL_x \bar{\oplus} PENABLE \Rightarrow \mathbf{EX} PSEL_y))$$

where $\bar{\oplus}$ is negated exclusive-OR. This property checks that $PSEL$ (for any

slave) can go high if the APB is idle or has just finished a transfer. The model checker returns the required theorems.

5.3 Verifying AHB

We verify arbitration, latency, coherence and deadlock freedom properties for AHB. The BDD variable ordering used was an interleaving of the current and next-state variables, which was then reordered after a manual dependency analysis.

5.3.1 Arbitration

The first properties we verify relate to arbitration. Typically such properties confirm that the arbiter is fair in some sense.

Our implementation is a simple priority based one and is obviously not meant to be fair in the sense that all requests are ultimately granted. This should hold true for the highest priority master m however. This can be checked using the CTL property

$$\mathbf{AG}(BUSREQ\ m \Rightarrow \mathbf{AXGRANT}\ m \vee \mathbf{AFHSPLIT}\ m \Rightarrow \mathbf{AXGRANT}\ m)$$

This states that the highest priority master is either granted immediately (if it was not masked) or is masked (which means it is waiting on a split transfer) and is granted when the split transfer it is waiting on is continued via an HSPLIT signal.

For other masters, the best we can hope for is that the possibility of a grant exists. This is the same property as before, except that m is replaced by the number of the master, and \mathbf{AX} is replaced by \mathbf{EX} throughout.

5.3.2 Latency

Latency checking for the AHB is more complicated than for the APB, as the presence of bursts, busy signals and wait states means that the transfer times are variable.

Since we have limits on the length of bursts, the number of consecutive busy signals and the number of consecutive wait states, we should be able to confirm that a transfer will take at most a given number of cycles. This number is in fact 34 cycles (1 address phase cycle + 16 burst cycles + 16 wait states + 1 BUSY signal) in the case of our implementation so far. The CTL property saying this is more neatly expressed if we first define a function LAT :

$$\begin{aligned} LAT\ f\ 0 &= f \\ LAT\ f\ (n + 1) &= f \vee \mathbf{AX}(LAT\ f\ n) \end{aligned}$$

This expresses in CTL a latency of at most n cycles until the event described by f holds. The required property is then given by the following CTL property:

$$\begin{aligned} \mathbf{AG}((NSQ \wedge SINGLE \Rightarrow LAT(READY \wedge OK) 2) \wedge \\ (NSQ \wedge INC \Rightarrow LAT((READY \wedge OK) \\ \vee RETRY \vee ERROR \vee SPLIT) 34) \wedge \\ \mathbf{AXA}[\neg NSQ \mathbf{U}(READY \wedge OK) \\ \vee RETRY \vee ERROR \vee SPLIT])) \end{aligned}$$

noting that a single transfer (signalled by SINGLE) takes only two cycles and that a burst (signalled by INC), if not interrupted, must finish within 34 cycles. An unfolding of LAT would reveal several relational product computations, which are time and space consuming. We can make our task easier by using the following lemma derived from the CTL semantics.

Lemma 5.2

$$\vdash \forall fgMs.s \models_M \mathbf{AG}(f \wedge g) \iff s \models_M \mathbf{AG}f \wedge s \models_M \mathbf{AG}g$$

Proof Simple rewriting with our formal semantics of L_μ and CTL [1]. \square
We can thus split⁴ the latency property above into the two conjuncts

$$\mathbf{AG}(NSQ \wedge SINGLE \Rightarrow LAT(READY \wedge OK) 2) \quad (4)$$

and

$$\begin{aligned} \mathbf{AG}(NSQ \wedge INC \Rightarrow LAT((READY \wedge OK) \\ \vee RETRY \vee ERROR \vee SPLIT) 10) \wedge \\ \mathbf{AXA}[\neg NSQ \mathbf{U}(READY \wedge OK) \\ \vee RETRY \vee ERROR \vee SPLIT]) \end{aligned} \quad (5)$$

Using standard propositional logic lifted to CTL (trivial using HOL) together with our formal CTL semantics, we can further split conjunct 5 above into

$$\begin{aligned} \mathbf{AG}(NSQ \wedge INC \Rightarrow LAT((READY \wedge OK) \\ \vee RETRY \vee ERROR \vee SPLIT) 10) \end{aligned} \quad (6)$$

and

$$\begin{aligned} \mathbf{AG}(NSQ \wedge INC \Rightarrow \mathbf{AXA}[\neg NSQ \mathbf{U}(READY \wedge OK) \\ \vee RETRY \vee ERROR \vee SPLIT]) \end{aligned} \quad (7)$$

Now the satisfiability theorem for conjunct 7 follows from a simplified version of the latency property which is model checked easily. The satisfiability theorems for conjuncts 4 and 6 are derived by model checking. All three resulting theorems can then be recombined in HOL using lemma 5.2 to give the required theorem.

5.3.3 Coherence

Coherence properties are verified practically identically to the manner in which they are done for APB. As before the datapath compression makes this feasi-

⁴ Technically of course, we are not splitting the formula but the statement of its satisfiability. We elide these details to avoid clutter.

ble.

5.4 Verifying AMBA

So far, we have separately checked correctness properties for the AHB and APB components of AMBA. Ideally, since the signals of the AHB and the APB do not overlap, these properties hold in the combined system, in which the APB is connected via a bridge to the AHB. However, conjoining R_{AHB} and R_{APB} will result in a large system which may be infeasible or time consuming to model check directly. We can instead construct a compositional proof in the theorem prover.

The first task is to define the bridge. This is the APB master that acts as a slave to the AHB. We first define the states over which the bridge would operate.

Definition 5.3 $\bar{s}_{bridge} = \bar{s}_{AHB} \times \bar{s}_{APB}$

and as before we write \bar{s}'_{bridge} to denote the “next” state. The bridge transition relation R_{bridge} follows from this.

Definition 5.4

$$R_{bridge}(\bar{s}_{bridge}, \bar{s}'_{bridge}) = R_{AHB}^{slave_x}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \wedge R_{APB}^{master}(\bar{s}_{APB}, \bar{s}'_{APB})$$

Now we can define a new transition relation for the APB with R_{bridge} as the master. We shall call this transition relation R_{APB2} .

Definition 5.5

$$R_{APB2}(\bar{s}_{APB}, \bar{s}'_{APB}) = (\exists \bar{s}_{AHB} \bar{s}'_{AHB}. R_{bridge}(\bar{s}_{bridge}, \bar{s}'_{bridge})) \wedge R_{APB}^{slave}(\bar{s}_{APB}, \bar{s}'_{APB})$$

We use existential abstraction to hide behaviours we wish to ignore. This allows us to show that the new transition relation preserves all behaviours.

Lemma 5.6 $\vdash R_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) = R_{APB2}(\bar{s}_{APB}, \bar{s}'_{APB})$

Proof In the \Rightarrow direction we need to furnish the appropriate witnesses for the existentially quantified variables. This is done by using the integrated SAT solver in HOL to find a satisfying assignment for $R_{AHB}^{slave_x}(\bar{s}_{AHB}, \bar{s}'_{AHB})$. The rest follows by simplification. The \Leftarrow direction is straightforward. \square .

Using Lemma 5.6, it is trivial to show that the properties proved in the model M_{APB} with transition relation R_{APB} also hold in the model M_{APB2} with transition relation R_{APB2} .

Theorem 5.7 $\vdash \forall f. \bar{s}_{APB} \models_{M_{APB}} f \Rightarrow \bar{s}_{APB} \models_{M_{APB2}} f$

We can similarly define R_{AHB2} in which we can replace one of the generic slaves with R_{bridge} , this time hiding the APB signals, and conclude that all properties proved for the *AHB* hold when one of the slaves is the *APB* master.

At a more general level, we can show, without any extra model checking, that properties proved for AHB and APB hold in the combined system. First we need a technical lemma.

Lemma 5.8 *If any M_1 and M_2 are the same except that $M_1.AP \subseteq M_2.AP$, then*

$$\forall f s_1 s_2. s_1 \models_{M_1} f \iff s_2 \models_{M_2} f$$

Here, $M_1.AP$ represents the set of atomic propositions occurring in the definition of M_1 . This just states that adding extra unused propositions to a model does not change its behaviour. Note that the underlying state type of the two models is different and thus trivial amendments have to be made to M_2 to satisfy the type checker. The main result then states that properties proved for a sub-system can be shown to be true of the entire system, provided certain conditions hold.

Theorem 5.9 *For any universal property f and models M_1 and M_2 ,*

$$\forall s. s \models_{M_1} f \Rightarrow s \models_{M_2} f$$

provided M_2 is the synchronous parallel composition of M_1 with some M such that every transition of M_2 has a corresponding transition in M_1 .

Note that Theorem 5.9 requires both models to have the same state type. This is where Lemma 5.8 is used (to add the extra propositions of the system M_2 to the sub-system M_1).

We can now define the full AMBA model M_{AMBA} by defining

$$R_{AMBA} = R_{AHB2} \wedge R_{APB2}$$

and defining the rest of the M_{AMBA} tuple in the usual manner. Then, for example, we can take M_{APB} as M_1 and M_{AMBA} as M_2 , and use Theorem 5.7 and Theorem 5.9 to show that all universal APB properties hold in the AMBA system. And similarly for the AHB. We have thus proved, without using the model checker, that all universal properties proved for AHB and APB separately also hold in the combined system. This result does not apply to the non-universal deadlock freedom properties; deadlock freedom in a sub-system does not imply deadlock freedom overall.

Though we used interactive theorem proving, the general technique can be applied in any similar situation and it is possible to envision writing proof script generation functions in ML that would automate much of the task.

6 Related Work

Two recent verifications targeting AMBA AHB were presented in 2003. The first work [12] uses the ACL2 theorem prover to prove arbitration and coherence properties for arbitrary numbers of slaves and masters. Time is abstracted

away and intra-transfer complications (such as bursts, wait states, splits and retries) are ignored. This is because theorem provers used alone are better suited for attacking datapath properties at a high level of abstraction, without the clutter of cycle-level control signals.

The second work uses the SMV model checker to fix bugs in an academic implementation (they ignore the datapath and several control signals and use the minimal number of masters and slaves) of AMBA AHB [11]. They concentrate on a no-starvation violation (a master is denied access to the bus forever) which however is caused by an error in the implementation of their arbiter rather than in the protocol itself.

More recently, work is in progress on porting a Z specification of AMBA AHB [10] to HOL. This work is still in the draft stage.

A recent Ph.D. thesis [13] verifies roughly the same set of AHB properties as ours for a more complex implementation using the CADENCE SMV model checker and the ACL2 theorem prover and imports the results in HOL as trusted theorems. The emphasis here is on using specialist tools as oracles for HOL and the verification process itself is not discussed at length.

The almost total lack of interaction between control and data in bus designs makes it relatively easy to do the kind of abstractions that model checkers are good at. Bus architectures and the somewhat related domain of cache coherence protocols have thus long been staples of model checking case studies [4,5,7,9].

7 Conclusion

The AMBA specification is a 110 page document, laying out the design in the usual mix of english and timing diagrams. We have developed a formal HOL model of the AHB and APB components at the cycle-level and model-checked standard properties. We have then used HOL to compose the two verifications. It remains to be investigated whether we can extend this approach to verify properties over arbitrary numbers of masters and slaves.

The complete high-level model had 172 control bits, and, effectively, an arbitrary number of datapath bits which were reduced to 206 bits via datapath compression. The model checking runs were not particularly time or space intensive and all went through in a few minutes at most on a 3.0GHz Pentium IV, using no more than about 350MB of RAM. We attribute this to our restricted AHB model, the decompositions, and datapath compression.

The work illustrates how we can seamlessly combine theorem proving, model checking and SAT solvers for abstract modelling, and to perform decomposition (e.g. the AHB latency theorem and Theorem 5.9) and abstraction (e.g. datapath compression and Theorem 5.7) for model checking. It is different from earlier work in that *both* control and data properties are verified for APB+AHB using a single integrated system, achieving a pragmatic balance between efficiency and soundness.

References

- [1] Amjad, H., *Programming a symbolic model checker in a fully expansive theorem prover*, in: D. A. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science **2758** (2003), pp. 171–187.
- [2] ARM Limited, “AMBA Specification,” 2.0 edition (1999), ©ARM Limited. All rights reserved.
- [3] Ben-Ari, M., Z. Manna and A. Pnueli, *The temporal logic of branching time*, Acta Informatica **20** (1983), pp. 207–226.
- [4] Campos, S., E.M. Clarke, W. Marrero and M. Minea, *Verifying the Performance of the PCI Local Bus using Symbolic Techniques*, in: A. Kuehlmann, editor, *Proceedings of the IEEE International Conference on Computer Design (ICCD '95)*, Austin, Texas, 1995.
- [5] Clarke, E. M., O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan and L. A. Ness, *Verification of the Futurebus+ cache coherence protocol*, Technical Report CMU-CS-92-206, Carnegie Mellon University (1992).
- [6] Clarke, E. M., O. Grumberg and D. Peled, “Model Checking,” The MIT Press, 1999.
- [7] Goel, A. and W. R. Lee, *Formal verification of an IBM CoreConnect processor local bus arbiter core*, in: G. De Micheli, editor, *37th Conference on Design Automation (DAC 2000)*, ACM, 2000.
- [8] Kozen, D., *Results on the propositional mu-calculus*, Theoretical Computer Science **27** (1983), pp. 333–354.
- [9] McMillan, K. L., *Parameterized verification of the FLASH cache coherence protocol by compositional model checking*, in: T. Margaria and T. F. Melham, editors, *Proceedings of the 11th International Conference on Correct Hardware Design and Verification Methods*, LNCS **2144** (2001), pp. 179–195.
- [10] Newey, M., *A Z specification of the AMBA high-performance bus* (2004), draft.
- [11] Roychoudhury, A., T. Mitra and S. R. Karri, *Using formal techniques to debug the AMBA system-on-chip bus protocol*, in: *Design, Automation and Test in Europe*, IEEE Computer Society, Munich, Germany, 2003, pp. 10828–10833.
- [12] Schmaltz, J. and D. Borrione, *Validation of a parameterized bus architecture using ACL2*, in: W. H. Jr., M. Kaufmann and J. Moore, editors, *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, Boulder CO, USA, 2003.
- [13] Susanto, K. W., “A Verification Platform for System on Chip,” Ph.D. thesis, Department of Computing Science, University of Glasgow, UK (2004), private copy.