Technical Report

UCAM-CL-TR-602
ISSN 1476-2986

Number 602

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Model checking the AMBA protocol in HOL

Hasan Amjad

September 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

Model checking the AMBA protocol in HOL

Hasan Amjad

Abstract

The Advanced Microcontroller Bus Architecture (AMBA) is an open System-on-Chip bus protocol for high-performance buses on low-power devices. In this report we implement a simple model of AMBA and use model checking and theorem proving to verify latency, arbitration, coherence and deadlock freedom properties of the implementation.

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 Architecture[1] (AMBA) is an open System-on-Chip bus protocol for high-performance buses on low-power devices. In this report we implement a simple model of AMBA and verify latency, arbitration, coherence and deadlock freedom properties of the implementation.

The verification is conducted using a model checker for the modal μ-calculus L_μ, that has been embedded in the HOL theorem prover [3]. 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 [4]. This tight connection between model checking and theorem proving is exploited in section 4 of this report.

1 AMBA Overview

The AMBA specification defines three buses:

- Advanced High-performance Bus (AHB): The AHB 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 multiplexor and an address decoder. Typical bus masters are processors and DMA devices.
- Advanced System Bus (ASB): The ASB is also a system bus that can be used as an alternative to the AHB when the high-performance features of AHB are not required.

[1] ©1999 ARM Limited. All rights reserved. AMBA is a trademark of ARM Limited.

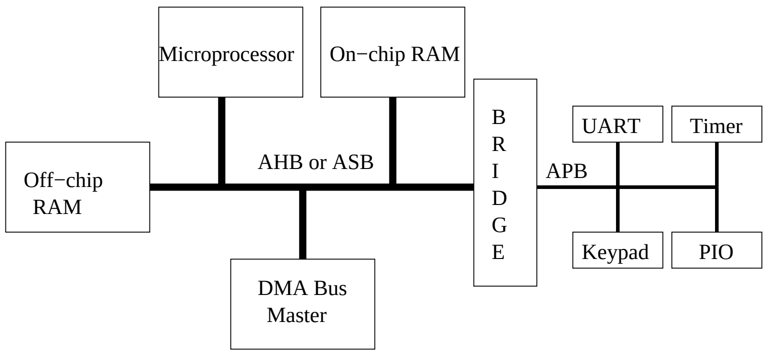

Figure 1: Typical AMBA-based Microcontroller

- 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.

Designers can use either the AHB or the ASB in conjunction with the APB. The APB has a single bus master module that acts as a bridge between the AHB or ASB 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 [5].

2 AMBA APB

The APB is optimized for low power consumption and low interface complexity. It is used for connecting the high-bandwith 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.

2.1 Specification

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

1. *IDLE*. This is the initial and the default state of the bus when no transfer is underway.

2. *SETUP*. The first stage of a transfer is a move to the SETUP state. The address, data and control signals are asserted during this phase but may not be stable. This stage always lasts for one clock cycle and then the operation moves to the ENABLE stage.

3. *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 depending on whether or not another transfer is required.

Table 1: AMBA APB Signals

Signal	Description
PCLK	The bus clock. The rising edge is used to trigger all APB signals.
PRESET	The reset signal. Resets the bus to the IDLE state. It is the only signal that is active low.
PSEL$_x$	This signal indicates that slave x is selected and a transfer is required, thus moving the bus from the IDLE to the SETUP stage. There is a unique line for each slave on the bus.
PENABLE	This signal triggers a move from the SETUP to the ENABLE stage, when the data and address buses are actually sampled.
PWRITE	When high this signal indicates a write access, when low a read access.
$PADDR$[31:0]	The address bus. Can be up to 32 bits wide and is driven by the bus master.
PRDATA[31:0]	The read data bus. It can be up to 32 bits wide and is driven by the selected slave (see PSEL$_x$) during a read access.
PWDATA[31:0]	The write data bus. It can be up to 32 bits wide and is driven by the bus master during a write access.

Table 1 lists all APB signals and their function. Each signal name is prefixed with P to denote that this is an APB signal.

2.1.1 Bus Master

There is a single bus master on the APB, thus there is no need for an arbiter. The master drives the address and write buses and also performs a combinatorial decode of the address to decide which $PSEL_x$ signal to activate. It is also responsible for driving the *PENABLE* signal to time the transfer. It drives APB data onto the system bus during a read transfer.

2.1.2 Slave

An APB slave drives the bus during read accesses. This can be done when the appropriate $PSEL_x$ is high and *PENABLE* goes high. $PADDR$ is used to determine the source register.

In a write transfer, it can sample write data at either the edge of $PCLK$ or *PENABLE*, when its $PSEL_x$ signal is high. Then $PADDR$ can be used to determine the target register.

2.2 Implementation

We implement the APB by following the specification in a straightforward manner without any optimizations. We need to implement the model as a state machine M_{APB}. In this case, a state is a tuple of all the signals considered as boolean variables. We use the standard convention of using primes to denote components of the target (or next) state in a transition.

Definition 1

$$\begin{aligned}\bar{s}_{APB} \;=\; & (PCLK, PRESET, PSEL_x, PENABLE, PWRITE,\\ & PADDR[31:0], PRDATA[31:0], PWDATA[31:0])\end{aligned}$$

and $\bar{s}'_{APB}$ represents $\bar{s}_{APB}$ with all components primed.

Note that $PSEL_x$ represents several variables, and $PADDR[31:0]$, $PRDATA[31:0]$ and $PWDATA[31:0]$ can each represent up to 32 variables. Henceforth, we will use this notational convention to abbreviate parameterised signals and address and data buses.

2.2.1 Assumptions

Some simplifying assumptions:

1. All signals are valid throughout, i.e. there is no glitching.

2. Sub-cycle timing (i.e. timing delays between signals becoming stable after changing) is ignored.

3. 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.

4. Endian-ness is not fixed, but is required to be consistent throughout.

5. We do not model reset as it is easy to do so but its presence trivially guarantees absence of deadlock.

These assumptions preserve the properties of the model that we are interested in.

2.2.2 The Model

We first need to define our state machine as a Kripke structure M_{APB}. S and L are defined in the obvious manner. M_{APB} is then described by an initial states predicate S_{0APB} on states, and a transition predicate R_{APB} which is a relation on states and is a conjunction – since the state machine is synchronous – of the transition relations for the components of the bus. As much as possible of the internal behaviour of the master and slaves has been abstracted.

The initial states predicate says simply that we start in the $IDLE$ stage.

Definition 2

$$S_{0APB}(\bar{s}_{APB}) = \bigwedge_x \neg PSEL_x \wedge \neg PENABLE$$

We need two transition relations, for the master and for slaves.

Definition 3

$$\begin{aligned}
& R_{APB}^{master}(\bar{s}_{APB}, \bar{s}'_{APB}) = \\
& (PENABLE' \iff PSEL_x \wedge \neg PENABLE) && (1)\\
\wedge\ & (PWRITE' \iff PSEL_x \Rightarrow PWRITE) && (2)\\
\wedge\ & (PADDR'_b \iff PSEL_x \Rightarrow PADDR_b) && (3)\\
\wedge\ & ((PSEL_x \iff \neg PENABLE) \Rightarrow PSEL'_x) && (4)\\
\wedge\ & (Mst'_{r,b} \iff \text{if}\,(\neg PWRITE \wedge ((r,b) = DECODE(PADDR)) \\
& \qquad \wedge PSEL_x \wedge PENABLE) \\
& \qquad \text{then}\, Slv_{x,r,b}\, \text{else}\, Mst_{r,b}) && (5)
\end{aligned}$$

Note that some of the transition conditions represent schema. $PADDR_b$ represents line b of the address bus, $Mst_{r,b}$ represents bit b of register r of the master, and $Slv_{x,r,b}$ represents bit b of register r of slave x, where the x is the same as the x in $PSEL_x$. We use *Mst* and *Slv* to model actual master and slave registers because it is easier to check coherency properties this way rather than by modelling the data buses $PRDATA$ and $PWDATA$, specially since we are ignoring glitching and sub-cycle timing.

Line 1 of Definition 3 drives *PENABLE* to high immediately after the cycle in which $PSEL_x$ goes high. Line 2 latches the value of $PWRITE$ once $PSEL_x$ is high. Line 3 does the same for $PADDR$. Line 4 ensures that $PSEL_x$ stays high in the ENABLE stage. Finally, line 5 ensures that the master regisers are updated correctly; the $DECODE$ function recovers which bit of which register of the master is to be updated.

Slaves have a very simple transition relation.

Definition 4

$$\begin{aligned}
& R_{APB}^{slave}(\bar{s}_{APB}, \bar{s}'_{APB}) = \\
& Slv'_{x,r,b} \iff \text{if}\,(PWRITE \wedge ((r,b) = DECODE(PADDR)) \\
& \qquad \wedge PSEL_x \wedge PENABLE) \\
& \qquad \text{then}\, Mst_{r,b}\, \text{else}\, Slv_{x,r,b}
\end{aligned}$$

The definition ensures that slave registers are updated correctly.

The APB transition relation is just the conjunction of the transition relations for the master and slave modules.

Definition 5

$$\begin{aligned}
& R_{APB}(\bar{s}_{APB}, \bar{s}'_{APB}) = \\
& R_{APB}^{master}(\bar{s}_{APB}, \bar{s}'_{APB}) \wedge R_{APB}^{slave}(\bar{s}_{APB}, \bar{s}'_{APB})
\end{aligned}$$

2.3 Verification

We verify three types of properties for our APB implementation. In all cases, a property is considered verified if the set of satisfying states include the initial states. This condition can be built into the properties but we do not do so to avoid computing the set of initial states repeatedly.

2.3.1 Latency

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. In our case, we want to confirm that all transactions take precisely two cycles.

Unfortunately this property cannot be represented in CTL , since it needs to be of the schematic form

$$\mathbf{AG}(\mathbf{A}(\mathbf{X}(\neg \mathit{PENABLE} \wedge \mathit{PSEL}_x) \Rightarrow \mathbf{XAXAX}(\mathit{PSEL}_x \wedge \mathit{PENABLE})))$$

which is a CTL * property.

We have not yet implemented a translation from CTL * to the propositional μ-calculus L_μ, so we are unable to check this property since our model checker [3] works for L_μ properties. The best we can do with CTL is the property schema

$$\mathbf{AG}(\neg \mathit{PENABLE} \wedge \mathit{PSEL}_x \Rightarrow \mathbf{AX}(\mathit{PSEL}_x \wedge \mathit{PENABLE}))$$

which checks that once a transfer starts, it finishes in the next cycle. Running this through the model checker returns the required theorem.

Theorem 6

$$\begin{aligned}&\vdash \forall \bar{s}_{APB}.\\ &\qquad \bar{s}_{APB} \models_{M_{APB}} \mathbf{AG}(\neg \mathit{PENABLE} \wedge \mathit{PSEL}_x\\ &\qquad\qquad\qquad\qquad \Rightarrow \mathbf{AX}(\mathit{PSEL}_x \wedge \mathit{PENABLE}))\end{aligned}$$

Theorem 6 is actually a family of theorems indexed by x, since the property was stated as a schema. Each theorem in the family is model checked separately. This applies to all theorems in this chapter that correspond to property schema, though we shall refer to each family as a Theorem to preserve the correspondence with the the associated property.

We mention in passing that the model checker would actually have returned the theorem

$$\begin{aligned}&\vdash \forall \bar{s}_{APB}.\\ &\qquad \bar{s}_{APB} \models^{\perp}_{\mathcal{T}(M_{APB})} \mathcal{T}(\mathbf{AG}(\neg \mathit{PENABLE} \wedge \mathit{PSEL}_x\\ &\qquad\qquad\qquad\qquad \Rightarrow \mathbf{AX}(\mathit{PSEL}_x \wedge \mathit{PENABLE})))\end{aligned}$$

from which Theorem 6 is derived using our translation of CTL to L_μ [1]. This applies to all other theorems returned by the model checker where we checked for CTL properties.

We can express the CTL * property directly in L_μ but this approach is best avoided as we currently lack the safety net of a formal translation from CTL * and L_μ is fairly non-intuitive to work with. However, we are able to get around this problem in the next section.

2.3.2 Coherence

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 required CTL property schema is

$$\begin{array}{ll}\mathbf{AG} & \\ & ((\neg PENABLE \wedge PSEL_x \wedge PWRITE \\ & \quad \wedge ((r,b) = DECODE(PADDR)) \\ & \qquad\qquad \Rightarrow ((\mathbf{AXAX} Slv_{x,r,b}) \iff Mst_{r,b})) \\ \wedge & (\neg PENABLE \wedge PSEL_x \wedge \neg PWRITE \\ & \quad \wedge ((r,b) = DECODE(PADDR)) \\ & \qquad\qquad \Rightarrow (Slv_{x,r,b} \iff (\mathbf{AXAX} Mst_{r,b}))))\end{array}$$

in which we can check coherency and a two-cycle transfer time simultaneously. The two conjuncts check for coherency during write and read cycles respectively. The model checker returns the required theorem.

Theorem 7

$$\begin{array}{ll}\vdash & \forall \bar{s}_{APB}. \\ & \quad \bar{s}_{APB} \models_{M_{APB}} \\ \mathbf{AG} & \\ & ((\neg PENABLE \wedge PSEL_x \wedge PWRITE \\ & \quad \wedge ((r,b) = DECODE(PADDR)) \\ & \qquad\qquad \Rightarrow ((\mathbf{AXAX} Slv_{x,r,b}) \iff Mst_{r,b})) \\ \wedge & (\neg PENABLE \wedge PSEL_x \wedge \neg PWRITE \\ & \quad \wedge ((r,b) = DECODE(PADDR)) \\ & \qquad\qquad \Rightarrow (Slv_{x,r,b} \iff (\mathbf{AXAX} Mst_{r,b}))))\end{array}$$

2.3.3 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. Since our transition relation has been defined by assigning all next-state variables some value in each cycle, the simple CTL property

$$\mathbf{AG}\,\mathbf{EX}\, True$$

(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 6. 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.

Theorem 8

$$\begin{aligned} \vdash\ & \forall \bar{s}_{APB}. \\ & \bar{s}_{APB} \models_{M_{APB}} \mathbf{AG}(\mathbf{AF}(PSEL_x \bar{\oplus} PENABLE \Rightarrow \mathbf{EX}\, PSEL_y)) \end{aligned}$$

3 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 choose to model the AHB rather than the ASB because the AHB is a newer design and also because it has been designed to integrate well with the verification and testing workflow.

3.1 Specification

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):

1. The AHB is in the default or initial state. No transfer is taking place, all slaves are ready and no master requires a transfer.

2. Several masters request the bus for a transfer.

3. The arbiter grants the bus according to some priority-scheduling algorithm.

4. The granted master puts the address and control information on the bus.

5. The decoder does a combinatorial decode of the address and the selected slave samples the address.

6. 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:

- The master or the slave may extend the transfer by inserting idle cycles or wait states during the transfer.

- The master may indicate a burst in which case several transfers occur end-to-end.

Table 2: AMBA AHB Master Signals

Signal	Driver	Description
HADDR[31:0]	Master	The address bus. Up to 32 bits wide.
HTRANS[1:0]	Master	Indicates the type of the current transfer. These can be idle (IDLE), busy (BUSY), non-sequential (NSQ) or sequential (SEQ).
HWRITE	Master	When high this indicates a write transfer (master to slave) and a read when low.
HBURST	Master	Indicates if the transfer forms part of a burst.
HWDATA[127-31:0]	Master	The write data bus. Can be up to 128 bits wide.
HBUSREQ$_x$	Master	When high indicates to the arbiter that master x is requesting the bus. There is a separate bus request line for each master.

Table 3: AMBA AHB Slave Signals

Signal	Driver	Description
HRDATA[127-31:0]	Slave	The read data bus. Can be up to 128 bits wide.
HSPLIT$_x$[15:0]	Slave	This is used by a slave x to tell the arbiter which masters should be allowed to re-attempt a split transfer. Each bit corresponds to a single master.
$HREADY$	Slave	When high indicates that a transfer is complete. Slaves can drive this low to insert wait states.
HRESP[1:0]	Slave	Allows the slave to provide additional information about a transfer. The reponses are okay (OK), error (ERR), retry (RETRY) and split (SPLIT).

- The slave may report an error and abort the transfer.
- The slave may signal a split or a retry, indicating it cannot at the moment proceed with the transfer. In this case the master may relinquish the bus and complete the transfer later (split) or not leave the bus and complete the transfer once the slave is ready (retry).

Tables 2, 3 and 4 list the AHB master, slave and other signals respectively and their function. Each signal name is prefixed with H to denote that this is an AHB signal.

3.1.1 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 $HBUSREQ_x$ to low and if it owns the bus it also drives $HTRANS$ to IDLE. To initiate a transfer, it drives $HBUSREQ_x$ high. Upon getting bus ownership (checked via $HGRANT_x$, $HMASTER$ and $HREADY$), the address and control signals are driven onto the bus for exactly one

Table 4: AMBA AHB System, Arbiter and Decoder Signals

Signal	Driver	Description
HCLK	Clock	The bus clock. The rising edge is used to trigger all AHB signals.
HRESET	System	The reset signal. Resets the bus to the default state. It is the only signal that is active low.
HGRANT$_x$	Arbiter	When high indicates that master x currently has the highest priority for getting the bus. Bus ownership does not actually change till the current transfer ends.
HMASTER[3:0]	Arbiter	Indicates which master is currently performing a transfer (and thus has the bus). Its timing is aligned with the address phase of the transfer. Used by SPLIT-enabled slaves.
HSEL$_x$	Decoder	This signal indicates that slave x is selected for the current transfer. There is a unique line for each slave on the bus. This signal is arrived at by decoding the higher order bits of the address bus.

cycle. To initiate the transfer, the master drives $HTRANS$ to NSQ (which abbreviates "non-sequential"). It also drives $HBURST$ to low indicating a single transfer, or to high indicating a four-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 increment the address signals to prepare for the next beat of the burst. In the middle of a burst $HTRANS$ is driven to SEQ.

It should be noted that in the last beat of a burst (or the one-cycle data phase of a single transfer) the information on the control and address buses is driven by the master that next has control of the bus, or by a *default master* (usually the highest priority master) if no master wishes to acquire the bus. In the latter case, the default master can simply drive $HTRANS$ to IDLE in which case the other signals are ignored. We will assume that Master 1 is the default master. Master 0 is reserved as a *dummy master* which guarantees to generate only IDLE transfers, and is granted the bus if all other masters are waiting on SPLIT transactions.

Responses to Slave Signals Masters need to respond to the following slave signals:

- If the slave drives $HREADY$ to low, then the master must continue the assert the same control, address and data signals in the next cycle, and continue this until $HREADY$ is high again.

- If the slave drives $HRESP$ to ERR, the master may abort the transfer or continue with it.

- If the slave drives $HRESP$ to SPLIT, the arbiter will grant the bus to another master. In this case the first master waits until it is given the bus again. The bus protocol only allows for masters to have one outstanding SPLIT transfer. Thus upon regaining the bus the master can continue with the transfer as before. However, a slave need not remember the control and address information and the master should broadcast this information first before driving/sampling the data buses.

- If the slave drives $HRESP$ to RETRY, the master simply retries the transfer until it is completed, which is indicated by the slave signalling OK. To prevent deadlock, only one master can access a slave that has issued the RETRY reponse.

3.1.2 Multiplexor

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

3.1.3 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 (via $HSPLIT_x$) that is it ready to continue the transfer.

3.1.4 Decoder

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

3.1.5 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 driving $HREADY$ to high and $HRESP$ to 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 $HREADY$ to low and $HRESP$ to 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. SPLIT-capable slaves need to be able to record the numbers of up to 16 masters to prevent deadlock. When ready, they activate the appropriate bits on $HSPLIT_x$ to indicate which master(s) the slave is ready to communicate with and continue with the transfers.

- If a non-SPLIT-capable slave cannot complete a transfer immediately it drives $HRESP$ to RETRY. To prevent deadlock, it must record the number of the current master and ensure that an ensuing transfer is with the same master, until the RETRY'd transfer is complete. If the master is not the same, the slave has the option of issuing an ERR, generating a system level interrupt or a complete reset.
- In case of a complete failure, the slave drives $HRESP$ to ERR, and ignores the rest of the transfer.

The RETRY, SPLIT and ERROR responses take two cycles (HREADY 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.

3.2 Implementation

We implement the AHB by following the specification in a straightforward manner without any optimizations. We need to implement the model as a state machine M_{AHB}, representing a state of M_{AHB} by $\bar{s}_{AHB}$.

Definition 9

$$\begin{aligned}\bar{s}_{AHB} \;=\; & (HTRANS[1:0], HREADY, HRESP[1:0], HSPLIT_x[15:0],\\ & HGRANT_x, HBUSREQ_x, HSEL_x, HADDR[31:0],\\ & HMASTER[1:0], HBURST, HWS_x, BB_x, HMASK_x,\\ & HSLVSPLIT_x)\end{aligned}$$

We write $\bar{s}'_{AHB}$ to represent $\bar{s}_{AHB}$ with all components primed.

The HWS_x, BB_x, $HMASK_x$ and $HSLVSPLIT_x$ signals are not part of the specification but are required by the implementation to count elapsed wait states and burst beats, and for the arbiter and slaves' internal bookkeeping. We shall refer to HWS_x and BB_x as counters.

3.2.1 Assumptions and Limitations

All assumptions made in §2.2 hold. We have made some additional assumptions to simplify the implementation a little.

Most importantly, we have not implemented the datapath. Datapath implementation and verification has already been demonstrated in §2 and verifying datapath properties for AHB is presently beyond the capabilities of our under-development model checker. The interesting aspects of the AHB all lie in the control circuitry. Other assumptions are:

- All bursts are four beats long. This encompasses all possible interactions that would be added by considering longer bursts.
- All bursts align at word boundries. Having non-aligned data does not affect the logical behaviour of the system but would increase the time to implement a working model. In fact, we restrict transfer size to be of word length.

- Slaves can insert up to four wait states. The specification leaves the actual number up to the implementer, but recommends no more than 16.
- We implement only three masters and two slaves. Again, this is the minimum number that encompasses all possible interactions and was considered sufficient for the purposes of this case study. With no datapath, the current system should scale up to the maximum easily without increasing the difficulty of model checking.

We have also not implemented some aspects of the specification (these and any signals they use have been left out of §3.1):

- Protection mechanisms are left out. These are given as optional in the specification.
- Locked bus access is left out.

3.2.2 The Model

As before, M_{AHB} is then described by an initial states predicate S_{0AHB} on states, and a transition predicate R_{AHB}. Due to the added complexity in the AHB, we define initial and transition predicates for the arbiter, decoder, multiplexor, masters, slaves and counters separately and take their conjunction to give the predicates for the system as a whole.

To improve readibility, we first define some predicates that abbreviate commonly used signal combinations:

Definition 10 *Abbreviations:*

- *Transfer types*
    1. *Idle:* $IDLE(HTRANS[1:0]) = \neg HTRANS_0 \wedge \neg HTRANS_1$
    2. *Busy:* $BUSY(HTRANS[1:0]) = HTRANS_0 \wedge \neg HTRANS_1$
    3. *Non-seq:* $NSQ(HTRANS[1:0]) = \neg HTRANS_0 \wedge HTRANS_1$
    4. *Sequential:* $SEQ(HTRANS[1:0]) = HTRANS_0 \wedge HTRANS_1$
- *Slave reponses*
    1. *Okay:* $OK(HRESP[1:0]) = \neg HRESP_0 \wedge \neg HRESP_1$
    2. *Error:* $ERR(HRESP[1:0]) = HRESP_0 \wedge \neg HRESP_1$
    3. *Retry:* $RETRY(HRESP[1:0]) = \neg HRESP_0 \wedge HRESP_1$
    4. *Split:* $SPLIT(HRESP[1:0]) = HRESP_0 \wedge HRESP_1$
- *Burst types*
    1. *Single transfer:* $SINGLE(HBURST) = \neg HBURST$
    2. *4-beat incrementing burst:* $INC4(HBURST) = HBURST$

To further avoid clutter, we will elide the arguments to the abbreviation predicates. Thus $IDLE$ stands for $IDLE(HTRANS[1:0])$. Of course this elision is not carried out in the theorem prover itself since that would cause a typing error. Also, we will prime the abbreviation name to denote the priming of the signals it is defined over.

We now define the initial state predicates:

Definition 11 *Initial state predicates are defined as follows :*

- *Arbiter*

$$S_{0_{AHB}}^{arbiter}(\bar{s}_{AHB}) = \bigwedge_{x \neq 1} \neg HGRANT_x \wedge HGRANT_1 \wedge HMASTER = 1$$

- *Decoder*

$$S_{0_{AHB}}^{decoder}(\bar{s}_{AHB}) = \bigwedge_{x} \neg HSEL_x$$

- *Counters*

$$S_{0_{AHB}}^{counters}(\bar{s}_{AHB}) = \bigwedge_{x} \neg HWS_x \wedge \bigwedge_{x} \neg BB_x$$

- *Masters*

$$S_{0_{AHB}}^{master}(\bar{s}_{AHB}) = IDLE \wedge \bigwedge_{x \neq 1} \neg HBUSREQ_x \wedge HBUSREQ_1$$

- *Slaves*

$$S_{0_{AHB}}^{slave}(\bar{s}_{AHB}) = HREADY \wedge OKAY$$

These defaults are those recommended by the specification document [5]. The notational abuse $HMASTER = 1$ above simply means that the bits of $HMASTER$ are set to the binary representation of 1, under the given endianness.

The system initial state predicate is simply the conjunction.

Definition 12

$$\begin{aligned} S_{0_{AHB}}(\bar{s}_{AHB}) \quad = \quad & S_{0_{AHB}}^{counters}(\bar{s}_{AHB}) \wedge S_{0_{AHB}}^{arbiter}(\bar{s}_{AHB}) \\ & \wedge S_{0_{AHB}}^{decoder}(\bar{s}_{AHB}) \wedge S_{0_{AHB}}^{master}(\bar{s}_{AHB}) \\ & \wedge S_{0_{AHB}}^{slave}(\bar{s}_{AHB}) \end{aligned}$$

The transition predicates are more complicated:

Definition 13 *Arbiter transitions:*

$$
\begin{array}{l}
R^{arbiter}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
(HGRANT'_0 \iff (HMASK_0 \wedge HMASK_1)) \wedge \\
(HGRANT'_1 \iff (\text{if}\, HMASK_0 \,\text{then}\, \mathtt{F} \text{else}\, HBUSREQ_1) \vee \\
\qquad\qquad\qquad \neg(\text{if}\, HMASK_1 \,\text{then}\, \mathtt{F} \text{else}\, HBUSREQ_2)) \wedge \\
(HGRANT'_2 \iff \neg(\text{if}\, HMASK_0 \,\text{then}\, \mathtt{F} \text{else}\, HBUSREQ_1) \wedge \\
\qquad\qquad\qquad (\text{if}\, HMASK_1 \,\text{then}\, \mathtt{F} \text{else}\, HBUSREQ_2)) \wedge \\
(HMASTER' \iff \text{if}\, \neg HREADY \,\text{then}\, HMASTER \,\text{else}\, \neg HGRANT_1 \\
(HMASK'_x \iff \text{if}\, SPLIT \wedge \neg HREADY \wedge (HMASTER = x) \,\text{then}\, \mathtt{T} \\
\qquad\qquad\qquad \text{else if}\, HSPLIT_x \,\text{then}\, \mathtt{F} \text{else}\, HMASK_x)
\end{array}
$$

Recall we are implementing three masters only. The dummy master 0 gets granted if and only if both the other masters are waiting on split transfers. Master 1 which is the default master gets priority over Master 2, but bus requests are masked for masters waiting on split transfers. A grant by itself does not give bus ownership. This happens when $HREADY$ is high. $HMASTER$ then indicates who has the bus, according to the value of $HGRANT_x$.

Definition 14 *Decoder transitions:*

$$
\begin{array}{l}
R^{decoder}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
(HSEL'_0 \iff \text{if}\, HREADY \,\text{then}\, \neg HADDR_0 \,\text{else}\, HSEL_0) \wedge \\
(HSEL'_1 \iff \text{if}\, HREADY \,\text{then}\, HADDR_0 \,\text{else}\, HSEL_1)
\end{array}
$$

The decoder simply does a combinatorial decode of the higher order bits of the address bus. Since we have only two slaves and no datapath, a single-bit address bus suffices.

Definition 15 *Counter transitions:*

$$
\begin{array}{l}
R^{counter}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
(HWS'_0 \iff \neg HREADY) \wedge \\
(HWS'_1 \iff \neg HREADY \wedge HWS_0) \wedge \\
(HWS'_2 \iff \neg HREADY \wedge HWS_1) \wedge \\
(bb0' \iff HREADY \wedge NSQ) \wedge \\
(bb1' \iff \neg BB_2 \wedge SEQ \wedge \,\text{if} HREADY \wedge \neg BUSY \,\text{then}\, bb_0 \,\text{else}\, bb_1) \wedge \\
(bb2' \iff \neg BB_2 \wedge SEQ \wedge \,\text{if} HREADY \wedge \neg BUSY \,\text{then}\, bb_1 \,\text{else}\, bb_2)
\end{array}
$$

The wait state counter simply counts up to 3 if $HREADY$ is low, resetting if $HREADY$ goes high. The burst counters count four beats when a burst starts (signalled by a NSQ followed by a SEQ transfer type). This is used by the arbiter to determine when it is safe to hand the bus over to another master (recall that in our implementation bursts may not be interrupted).

Definition 16 *Multiplexor transitions:*

$$\begin{aligned}
&R^{mux}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
&(HTRANS' \iff \text{if}\, \neg HREADY \,\text{then}\, HTRANS \\
&\qquad\qquad \text{else if}\, HGRANT_1 \,\text{then}\, HTRANS_{m_1} \\
&\qquad\qquad \text{else if}\, HGRANT_2 \,\text{then}\, HTRANS_{m_2} \\
&\qquad\qquad \text{else}\, HTRANS_{m_0}) \wedge \\
&(HBURST' \iff \text{if}\, \neg HREADY \,\text{then}\, HBURST \\
&\qquad\qquad \text{else if}\, HGRANT_1 \,\text{then}\, HBURST_{m_1} \\
&\qquad\qquad \text{else if}\, HGRANT_2 \,\text{then}\, HBURST_{m_2} \\
&\qquad\qquad \text{else}\, HBURST_{m_0})
\end{aligned}$$

Drives the selected master control signals (i.e. $HTRANS_{m_x}$ etc) to the bus.

Definition 17 *Master transitions:*

$$\begin{aligned}
&R^{master_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
&(\neg(OK \wedge HREADY) \wedge HGRANT_x \Rightarrow HBUSREQ'_x) \wedge \\
&(HREADY \wedge IDLE \wedge OK \wedge HGRANT_x \Rightarrow NSQ') \wedge \\
&(NSQ \wedge OK \wedge INC4 \wedge HGRANT_x \Rightarrow (BUSY' \vee SEQ') \wedge INC4') \wedge \\
&(SEQ \wedge \neg BB_2 \wedge OK \wedge HGRANT_x \Rightarrow (BUSY' \vee SEQ') \wedge INC4') \wedge \\
&(BUSY \wedge \neg BB_2 \wedge OK \wedge HGRANT_x \Rightarrow SEQ' \wedge INC4') \wedge \\
&(\neg HREADY \wedge RETRY \wedge HGRANT_1 \Rightarrow IDLE') \wedge \\
&(HREADY \wedge RETRY \wedge HGRANT_1 \Rightarrow NSQ') \wedge \\
&(ERROR \wedge HGRANT_1 \Rightarrow IDLE')
\end{aligned}$$

A line-by-line explanation of this transition relation follows: if master has bus ownership it will continue to request it until the transfer completes, otherwise the arbiter may think the master no longer requires the bus in the middle of a transfer; starting a transfer by asserting NSQ; switching to the SEQ transfer signal if transfer is a burst, i.e. INC4 is being asserted; continuing to assert SEQ or BUSY as burst takes place; forcing a SEQ assert if BUSY was asserted previously (the specification does not mention this but it is clearly required to prevent an infinite sequence of BUSYs, i.e. a livelock); response to first and second cycle of retry; response to first and second cycle of error. According to the specification, the master can do whatever it likes if split, since it loses the bus.

The one exception to the above is Master 0, the dummy master. This simply generates IDLE no matter what happens, and never requests the bus.

Definition 18 *Slave transitions:*

$$\begin{array}{l}
R^{slave_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) = \\
(HSEL_x \wedge HWS_2 \Rightarrow HREADY') \wedge \\
(HSEL_x \wedge (NSQ \vee SEQ) \wedge OK \Rightarrow OK' \wedge HREADY') \wedge \\
(HSEL_x \wedge IDLE \Rightarrow HREADY' \wedge OK') \wedge \\
(HSEL_x \wedge BUSY \Rightarrow OK' \\
(HSEL_x \wedge \neg HREADY \wedge \neg(IDLE \vee BUSY) \wedge RETRY \\
\qquad\qquad\qquad \Rightarrow HREADY' \wedge RETRY') \wedge \\
(HSEL_x \wedge \neg HREADY \wedge \neg(IDLE \vee BUSY) \wedge ERROR \\
\qquad\qquad\qquad \Rightarrow HREADY' \wedge ERROR') \wedge \\
(HSEL_x \Rightarrow \neg SPLIT')
\end{array}$$

A line-by-line explanation: this line together with the wait state counter ensures that $HREADY$ never stays low for more than four consecutive cycles, enforcing the rule that slaves may not insert more than four wait states; signal end of transfer by asserting $HREADY$ and OK; reponse to IDLE signal is $HREADY$ and OK; response to BUSY signal is OK; drive second cycle of RETRY; drive second cycle of ERROR; do not ever signal SPLIT.

A SPLIT-capable slave is slightly more complex. To add SPLIT ability, we conjoin the above transition relation (excepting the last line) with the following.

$$\begin{array}{l}
(HSLVSPLIT'_x \iff \\
\qquad\qquad \text{if}\,(HSEL_x \wedge \neg HREADY \wedge SPLIT \wedge (HMASTER = x)) \\
\qquad\qquad \text{then}\,HMASTER_x\,\text{else}\,HSLVSPLIT_x) \wedge \\
(HSEL_x \wedge HGRANT_0 \Rightarrow OK) \wedge \\
(HSLVSPLIT_x \wedge (y \neq x) \Rightarrow \neg HSLVSPLIT_y)
\end{array}$$

The first conjunct is for recording the current bus master's number so when the slave is later ready it can assert the appropriate $HSPLIT_x$ line. We abstract as much of the behaviour as possible, but the next two conjuncts are required to prevent undesirable behaviour. The first disables splits if the dummy master has the bus, and the last ensures that the slave does not split if it has already done so. The specification recommends that slaves should be able to split on as many masters as are present. However, this simplification does not affect logical behaviour, only efficiency.

The conjunction gives the system transition relation:

Definition 19

$$\begin{array}{rcl}
R_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) & = & R^{counters}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \wedge R^{arbiter}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \\
& & \wedge\, R^{decoder}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \wedge R^{master_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB}) \\
& & \wedge\, R^{slave_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB})
\end{array}$$

3.3 Verification

We verify arbitration, latency and deadlock freedom properties for AHB. As there is no datapath we do not verify coherence. The BDD variable ordering used was an interleaving of the current and next-state variables, which was then reordered after a manual dependency analysis.

3.3.1 Arbitration

The first properties we verify relate to arbitration. Typically such properties confirm that the arbiter is fair in some sense. The first property we verify is mutual exclusion, i.e. two masters never simultaneously get granted. The CTL property for this is

$$\mathbf{AG}(HGRANT_x \wedge (x \neq y) \Rightarrow \neg HGRANT_y)$$

The required theorem is given by running the model checker.

Theorem 20

$$\begin{aligned}\vdash\ & \forall \bar{s}_{AHB}. \\ & \bar{s}_{AHB} \models_{M_{AHB}} \mathbf{AG}(HGRANT_x \wedge (x \neq y) \Rightarrow \neg HGRANT_y)\end{aligned}$$

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 1 however. This can be checked using the CTL property

$$\mathbf{AG}(HBUSREQ_1 \wedge \neg HMASK_1 \Rightarrow \mathbf{AF} HGRANT_1)$$

Note that a grant is not the same as getting bus ownership (Master 1 may de-assert its request while waiting for the bus). Thus this property holds and the model checker gives the required theorem.

Theorem 21

$$\begin{aligned}\vdash\ & \forall \bar{s}_{AHB}. \\ & \bar{s}_{AHB} \models_{M_{AHB}} \mathbf{AG}(HBUSREQ_1 \wedge \neg HMASK_1 \Rightarrow \mathbf{AF} HGRANT_1)\end{aligned}$$

For other masters, the best we can hope for is that the possibility of a grant exists, as given by the CTL property schema

$$\mathbf{AG}(HBUSREQ_x \wedge \neg HMASK_x \Rightarrow \mathbf{EF} HGRANT_x)$$

and the model checker confirms that this is so.

Theorem 22

$$\begin{aligned}\vdash\ & \forall \bar{s}_{AHB}. \\ & \bar{s}_{AHB} \models_{M_{AHB}} \mathbf{AG}(HBUSREQ_x \wedge \neg HMASK_x \Rightarrow \mathbf{EF} HGRANT_x)\end{aligned}$$

3.3.2 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.

First, we do a quick sanity check to confirm that all transfers do indeed end, as given by this CTL property:

$$\begin{aligned}&\mathbf{AG}(NSQ \Rightarrow \mathbf{AXA}[\neg NSQ\,\mathbf{U}\,(\mathit{HREADY} \wedge OK)\\ &\qquad\qquad\qquad\qquad \vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT}])\end{aligned}$$

and this is easily checked:

Theorem 23

$$\begin{aligned}\vdash\ &\forall \bar{s}_{AHB}.\\ &\bar{s}_{AHB} \vDash_{M_{AHB}} \mathbf{AG}(NSQ \Rightarrow \mathbf{AX}\,\mathbf{A}[\neg NSQ\mathbf{U}(\mathit{HREADY} \wedge OK)\\ &\qquad\qquad\qquad\qquad \vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT}])\end{aligned}$$

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 ten cycles (1 address phase cycle + 4 burst cycles + 4 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:

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

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:

$$\begin{aligned}\mathbf{AG}\quad &((NSQ \wedge SINGLE \Rightarrow LAT\,(\mathit{HREADY} \wedge OK)\,2) \wedge\\ &(NSQ \wedge INC4 \Rightarrow LAT\,((\mathit{HREADY} \wedge OK)\\ &\qquad\qquad \vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT})\,10 \wedge\\ &\qquad \mathbf{AXA}[\neg NSQ\,\mathbf{U}\,(\mathit{HREADY} \wedge OK)\\ &\qquad\qquad \vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT}]))\end{aligned}$$

noting that a single transfer takes only two cycles and that a burst, if not interrupted, must finish within ten 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 24

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

Proof Simple rewriting with our formal semantics of L_μ [3] and CTL [1]. □

We can thus split[2] the latency property above into the two conjuncts

$$\mathbf{AG} \quad (\mathit{NSQ} \wedge \mathit{SINGLE} \Rightarrow \mathit{LAT}\,(\mathit{HREADY} \wedge \mathit{OK})\,2) \tag{6}$$

and

$$\begin{aligned}\mathbf{AG} \quad (\mathit{NSQ} \wedge \mathit{INC4} \Rightarrow \mathit{LAT}\,((&\mathit{HREADY} \wedge \mathit{OK}) \\ &\vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT})\,10\, \wedge \\ \mathbf{AXA}[\neg \mathit{NSQ}\,\mathbf{U}\,(&\mathit{HREADY} \wedge \mathit{OK}) \\ &\vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT}])\end{aligned} \tag{7}$$

We then observe that the propositional fragment of L_μ has all the properties of normal propostional logic. In particular, we have

Lemma 25

$$\begin{aligned}&\forall M\, e\, s\, f_1\, f_2\, f_3\, f_4. \\ &s \models^e_M f_1 \wedge f_2 \Rightarrow f_3 \wedge f_4 \iff (f_1 \wedge f_2 \Rightarrow f_3) \wedge (f_1 \wedge f_2 \Rightarrow f_4)\end{aligned}$$

and

Lemma 26

$$\forall M\, e\, s\, f_1\, f_2\, f_3. s \models^e_M (f_1 \Rightarrow f_2) \Rightarrow f_1 \wedge f_3 \Rightarrow f_2$$

proved easily by the HOL simplifier using our formal semantics of L_μ [3].

Using Lemma 25 together with our formal CTL semantics we can further split conjunct 7 above into

$$\begin{aligned}\mathbf{AG} \quad (\mathit{NSQ} \wedge \mathit{INC4} \Rightarrow \mathit{LAT}\,((&\mathit{HREADY} \wedge \mathit{OK}) \\ &\vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT})\,10)\end{aligned} \tag{8}$$

and

$$\begin{aligned}\mathbf{AG} \quad (\mathit{NSQ} \wedge \mathit{INC4} \Rightarrow \mathbf{AXA}[\neg \mathit{NSQ}\,\mathbf{U}\,(&\mathit{HREADY} \wedge \mathit{OK}) \\ &\vee \mathit{RETRY} \vee \mathit{ERROR} \vee \mathit{SPLIT}])\end{aligned} \tag{9}$$

Now the satisfiability theorem for conjunct 9 follows from Theorem 23 using Lemma 26. The satisfiability theorems for conjuncts 6 and 8 are derived by model checking. All three resulting theorems can then be recombined in HOL using lemmas 24 and 25 to give the required theorem.

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

Theorem 27

$$\begin{array}{ll}\vdash & \forall \bar{s}_{AHB}.\\ & \bar{s}_{AHB} \models_{M_{AHB}} \mathbf{AG}((NSQ \wedge SINGLE \\ & \qquad\qquad \Rightarrow LAT\,(HREADY \wedge OK)\,2) \wedge \\ & \qquad (NSQ \wedge INC4 \\ & \qquad\qquad \Rightarrow LAT\,((HREADY \wedge OK) \\ & \qquad\qquad\qquad \vee RETRY \vee ERROR \vee SPLIT)\,10 \wedge \\ & \qquad\qquad \mathbf{AXA}[\neg NSQ\,\mathbf{U}\,(HREADY \wedge OK) \\ & \qquad\qquad\qquad \vee RETRY \vee ERROR \vee SPLIT]))\end{array}$$

Lemma 24 could also have been used in the derivation of Theorem 7 but in that case not much is gained by doing so as neither conjunct's evaluation results in large BDDs.

3.3.3 Deadlock Freedom

The transition relation for the AHB is not obviously total, unlike that for the APB. Thus the obvious way of checking for deadlock is the CTL property

$$\mathbf{AG}\,\mathbf{EX}\,True$$

Since CTL model checking requires the transition relation to be totalised, this property check needs to be carried out before totalisation. But then we cannot check for the CTL property.

Fortunately, due the fine-grained nature of our integration, we are not reliant on just getting a true/false answer from the model checker. We can simply "check" the property

$$\mathbf{EX}\,True$$

whose semantics are not affected by a non-totalised transition relation (only fix-point computations are affected), and then separately check whether the set of states returned by the model checker for the above property contains the set of reachable states $Reachable\,R_{AHB}\,S_{0AHB}$ of the system [4]. Thus we have the theorem

Theorem 28

$$\vdash Reachable\,R_{AHB}\,S_{0AHB} \subseteq \{s|s \models_{M_{AHB}} \mathbf{EX}\,True\}$$

which tells us that all reachable states have a next state and thus the system cannot deadlock. Subset inclusion here is modelled by propositional implication between the characteristic functions of the sets. The functions are boolean, so symbolic model checking can be used.

This property does not uncover situations where even though the transition system does not deadlock, it ends up in a useless loop doing nothing. To some extent, this is a liveness property and beyond the expressive power of CTL. We are considering how to best address this problem, either by writing L_μ properties or by finding a halfway solution that can be expressed in CTL.

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 29

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

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

Definition 30

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

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 31

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

As in our work on abstraction [2], 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 32

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

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 $R^{slave_x}_{AHB}(\bar{s}_{AHB}, \bar{s}'_{AHB})$. We know that such an assignment exists from Theorem 28, since the only way there is no satisfying assignment is if there are no transitions in the system. The rest follows by simplification. The $\Leftarrow$ direction is straightforward. □.

Using Lemma 32, 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 33

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

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 for AHB and APB hold in the combined system. First we need a technical lemma.

Lemma 34 *If any M_1 and M_2 are the same except that $M_1.AP \subseteq M_2.AP$, then*

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

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 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 35 *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 every behaviour of M_1 is a behaviour in M_2.

Note that Theorem 35 requires both models to have the same state type. This is where Lemma 34 is used (to add the extra propostions 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 33 and Theorem 35 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.

5 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 the bus. Time is abstracted away and intra-transfer complications (such as bursts, wait states, splits and retries) are ignored. This makes sense as theorem provers 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 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. The error is very subtle however and we concur with their conclusion that this particular case should be highlighted in the FAQ if not in the specification.

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 (it also verifies the datapath) for a more complex implementation using the CADENCE SMV model checker 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 complete 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 [6, 7, 8, 9].

6 Conclusion

The AMBA AHB and APB specification is a 110 page document, laying out the design in the usual mix of english, timing diagrams and interface diagrams, supplemented by a FAQ. We have developed a formal HOL version of the AHB and APB components at the cycle-level and model-checked useful properties. We have then used HOL to compose the two verifications.

However, while the case study is a useful show-case for our framework, there is much to be done for a complete verification. Priorities are verifying datapath properties for the AHB, implementing locked access, having a more sophisticated arbitration policy and non-word-aligned transfers.

During the case study it became clear that most of the time in a model checking oriented verification is spent patching failed properties or flawed models. Thus, the development of good failure analysis and debugging capabilites will go far in making the tool usable in practice.

The model checking runs were not particularly time or space intensive and all went through in a few minutes at most. We attribute this to our simplified model, the decomposition and abstraction we did, and our focus on control properties.

The case study illustrates how we can seamlessly combine theorem proving, model checking and SAT solvers to perform decomposition (e.g. Theorem 27 and Theorem 35) and abstraction (e.g. Theorem 33) for model checking. All steps are backed up by fully-expansive formal proof. We have thus enabled verifications that would be hard, if not infeasible, using only one of these technologies.

References

[1] H. Amjad. Formalizing the translation of CTL into L_μ. In David A. Basin and Burkhart Wolff, editors, *Supplementary Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, number 187 in Technical Reports, pages 207–217. Institut für Informatik, Albert-Ludwigs-Universität, 2003.

[2] H. Amjad. Implementing abstraction refinement for model checking in HOL. In David A. Basin and Burkhart Wolff, editors, *Supplementary Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, number 187 in Technical Reports, pages 219–228. Institut für Informatik, Albert-Ludwigs-Universität, 2003.

[3] H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In David A. Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2003.

[4] H. Amjad. Combining model checking and theorem proving. Technical Report UCAM-CL-TR-601, University of Cambridge Computer Laboratory, 2004. Ph. D. Thesis.

[5] ARM Limited. *AMBA Specification*, 2.0 edition, 1999. ©ARM Limited. All rights reserved.

[6] S. Campos, E.M. Clarke, W. Marrero, and M. Minea. Verifying the Performance of the PCI Local Bus using Symbolic Techniques. In Andreas Kuehlmann, editor, *Proceedings of the IEEE International Conference on Computer Design (ICCD '95)*, Austin, Texas, October 1995.

[7] E. M. Clarke, 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, October 1992.

[8] Amit Goel and William R. Lee. Formal verification of an IBM CoreConnect processor local bus arbiter core. In Giovanni De Micheli, editor, *37th Conference on Design Automation (DAC 2000).* ACM, June 2000.

[9] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the 11th International Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.

[10] Malcolm Newey. A Z specification of the AMBA high-performance bus. Draft, January 2004.

[11] Abhik Roychoudhury, Tulika Mitra, and S. R. Karri. Using formal techniques to debug the AMBA system-on-chip bus protocol. In *Design, Automation and Test in Europe*, pages 10828–10833, Munich, Germany, March 2003. IEEE Computer Society.

[12] Julien Schmaltz and Dominique Borrione. Validation of a parameterized bus architecture using ACL2. In Warren Hunt Jr., Matt Kaufmann, and J. Moore, editors, *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, Boulder CO, USA, July 2003.

[13] Kong Woei Susanto. *A Verification Platform for System on Chip.* PhD thesis, Department of Computing Science, University of Glasgow, UK, 2004. Private copy.