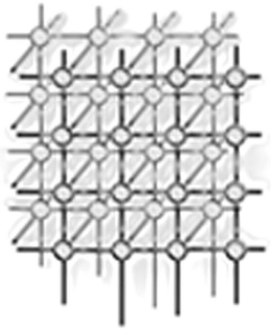
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **0**:1–12 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Modelling and analysis of the AMBA bus using CSP and B



Alistair A. McEwan*and Steve Schneider

Department of Engineering, University of Leicester, Leicester, LE2 7RH
Department of Computing, University of Surrey, Guildford, GU2 7XH

SUMMARY

In this paper, we present a formal model and analysis of the AMBA Advanced High-performance Bus. The model is given in CSP‖B—an integration of the process algebra CSP and the state-based formalism B. We describe the theory behind the integration of CSP and B, and present the model in this theory. Analysis is performed using the model-checker ProB. The contribution of this paper may be summarized as follows: presentation of a formal model of the AMBA AHB protocol such that it may be used for analysis of co-design systems incorporating the bus, an evaluation of the integration of CSP and B in the production of such a model, and a demonstration and evaluation of ProB in performing this analysis.

KEY WORDS: CSP‖B; AMBA; formal modeling; ProB; Co-Design

1. INTRODUCTION

In this paper we present a model of the AMBA Advanced High-performance Bus (AHB) in the formalism CSP‖B, and investigate this model using the model-checker and animator ProB. The AMBA bus, produced by ARM, is a freely available standard for on-chip busses in embedded systems. Implementations are available, and tools are available for the testing of components. We show that CSP‖B can be used to model the bus, and that models such as this can be used in the design, development, and formal analysis of hardware/software co-design systems. One goal of this work is to demonstrate how CSP‖B may be used in development, in a similar manner to that presented in [10]. It is our belief that the combination of the state based formalism B-Method, and the process algebra CSP permits accurate descriptions of the implementation of such systems that can be refined both to hardware and software; and the necessary potential for more abstract models for development and analysis purposes. The work in this paper was initiated under the AWE project "Future technologies for systems design".

*Correspondence to: alistair.mcewan@leicester.ac.uk

The paper begins in Section 2 by presenting background information on CSP‖B and ProB. Background on the AMBA bus, and the protocol of interest to this paper follows in Section 3. Section 4 presents the CSP‖B/ProB model. A discussion about the types of analysis that can be done on this model is presented in Section 5. The model, and the results of the analysis are different to our previous work, and the reasons for this are discussed in Section 6. Some conclusions are drawn in Section 7. The contribution of this paper can be summarized as follows: a demonstration of modeling components used in a typical co-design environment using CSP‖B, an evaluation of ProB in the development and analysis of CSP‖B modeling, and the presentation of an AMBA AHB model that can be used for the formal analysis and development of components to be attached to an implementation of the bus.

2. COMBINING CSP AND B

CSP‖B [20, 16] is a combination of the process algebra CSP [6, 13, 15] and the language of abstract machines supported by the B-Method [1, 14]. A *controlled component* consists of a B machine in parallel with a CSP controller process. Their interaction consists of synchronisations of B operations with corresponding events in the CSP controller. Consistency of the combination requires that operations are called only within their preconditions. Other properties of the combination may also be considered, such as deadlock-freedom, or various safety or liveness properties. Previous work has developed theory to verify controllers[17], and to combine them into larger systems[18]. The approach taken in this paper differs in that it applies a model-checker to the CSP‖B in order to achieve verification.

2.1. B Machines

The B-Method develops systems in terms of *machines*, which are components containing state and supporting operations on that state, described in a language called *Abstract Machine Notation* (AMN). An important aspect of B for this paper is that operations are associated with preconditions, and if called outside their preconditions then they diverge. A full description of the B-Method can be found in [1, 14], and tool support is provided by [4, 5].

A machine is defined using a number of clauses which each describe a different aspect of the machine. The MACHINE clause declares the abstract machine and gives its name. The VARIABLES clause declares the state variables used to carry the state information within the machine. The INVARIANT clause gives the type of the state variables, and more generally it also contains any other constraints on the allowable machine states. The INITIALISATION clause determines the initial state of the machine. The OPERATIONS clause contains the operations that the machine provides: these include query and update operations on the state.

2.2. CSP

CSP processes are defined in terms of the *events* that they can and cannot do. Processes interact by synchronising on events, and the occurrence of events is atomic. The set of all events is denoted by Σ. Events may be compound in structure, consisting of a *channel name*

and some (possibly none) *data values*. Thus, events have the form $c.v_1...v_n$, where c is the channel name associated with the event, and the v_i are data values. The *type* of the channel c is the set of values that can be associated with c to produce events. For instance, if *trans* is a channel name, and $\mathbb{N} \times \mathbb{Z}$ is its type, then events associated with *trans* will be of the form $trans.n.z$, where $n \in \mathbb{N}$ and $z \in \mathbb{Z}$. Therefore *trans*.3.8 would be one such event.

CSP has a number of semantic models associated with it. The most commonly used are the *Traces* model, and the *Failures/Divergences* model. Full details can be found in [13, 15]. A *trace* is a finite sequence of events. A sequence tr is a trace of a process P if there is some execution of P in which exactly that sequence of events is performed. The set $traces(P)$ is the set of all possible traces of process P. The traces model for CSP associates a set of traces with every CSP process. If $traces(P) = traces(Q)$ then P and Q are equivalent in the *traces model*, and we write $P =_T Q$. A *divergence* is a finite sequence of events tr. Such a sequence is a *divergence* of a process P if it is possible for P to perform an infinite sequence of internal events on some prefix of tr. The set of divergences of a process P is written $div(P)$. A *failure* is a pair (tr, X) consisting of a trace tr and a set of events X. It is a failure of a process P if either tr is a divergence of P (in which case X can be any set), or (tr, X) is a stable failure of P: a trace tr leading to a stable state in which no events of X are possible. The set of all possible failures of a process P is written $failures(P)$. If $div(P) = div(Q)$ and $failures(P) = failures(Q)$ then P and Q are equivalent in the *failures-divergences model*, written $P =_{FD} Q$.

Verification of CSP processes typically takes the form of refinement checking: where the behaviour of one process is entirely contained within the behaviour of another within a given semantic model. Tool support for this is offered by the model-checker FDR[9].

2.3. CSP semantics for B machines

Morgan's CSP-style semantics [12] for event systems enables the definition of such semantics for B machines. A machine M has a set of traces $traces(M)$, a set of failures $failures(M)$, and a set of divergences $div(M)$. A sequence of operations $\langle e_1, e_2 \dots e_n \rangle$ is a *trace* of M if it can possibly occur. This is true precisely when it is not guaranteed to be blocked, in other words it is not guaranteed to achieve *false*. In the *wp* notation of [12] this is $\neg wp(e_1;\ e_2;\ \dots;\ e_n, false)$, or in AMN $\neg([e_1;\ e_2;\ \dots;\ e_n]false)$. (The empty trace is treated as *skip*). A sequence does not diverge if it is guaranteed to terminate (i.e. establish *true*). Thus, a sequence is a divergence if it is *not* guaranteed to establish *true*, i.e. $\neg([e_1;\ e_2;\ \dots;\ e_n]true)$. Finally, given a set of events X, each event $e \in X$ is associated with a guard g_e. A sequence with a set of events is a *failure* of M if the sequence is not guaranteed to establish the disjunction of the guards. Thus, $(e_1;\ e_2;\ \dots;\ e_n, X)$ is a failure of M if $\neg[e_1;\ e_2;\ \dots;\ e_n](\bigvee_{e \in X} g_e)$. More details appear in [20]. The CSP semantics for B machines enables the combination of a B machine and a CSP process to be formally defined in terms of CSP semantics.

A *CSP controller* P is a process which has a given set of control channels (events) C. The controlled B machine will have a subset of $\{|\ C\ |\}$ as its alphabet, corresponding to the operations of the machine. Some of the controller events may not correspond to B operations. For each operation $w \longleftarrow e(v)$ of a controlled machine with v of type T_1 and w of type T_2 there will be a channel e of type $T_1 \times T_2$, so communications on e are of the form $e.v.w$. The operation call $e!v?x \rightarrow P$ is an interaction with an underlying B machine: the value v is

passed from the process as input to the B operation, and the value x is accepted as output from the B operation.

In previous work, controllers were generated from a sequential subset of CSP syntax[16], including prefixing, input, output, choice, and recursion. The motivation for this restriction was verification. Various consistency results were possible for combinations of B machines with such controllers by identifying control loop invariants which held at recursive calls. In this paper there is no need for such restrictions as we do not applying those techniques. Instead we use the ProB model-checker to establish results. This means that the full range of CSP syntax supported by ProB is available for expressing the CSP controllers.

2.4. ProB tool support

ProB [8] is an animator and model-checker for the B-Method. A B machine can be model-checked against its invariants, with counter-examples given when an invariant is violated. The latest version of ProB also includes support for a model incorporating a B machine and a CSP controller. The result is a combination of the two formalisms that is very similar in approach to CSP‖B. Although there are some differences to the way CSP‖B combines CSP and B, it is still a useful tool for developing, investigating, and animating CSP‖B models. In this paper we regard the combination of CSP and B as supported by ProB as the same as CSP‖B.

In [11], the version of CSP supported by ProB was not CSP_M and was therefore not FDR compatible. The latest version of ProB supports CSP_M, and therefore the controllers presented in this paper are compatible with FDR analysis.

3. THE AMBA BUS

The *Advanced Microcontroller Bus Architecture* (AMBA) is an on-chip communication standard for embedded micro controllers[3]. The standard is presented in an informal manner; and is intended to assist engineers connecting components to, or designing components for, the bus; and to support the modular development of complex *systems on a chip*. Freely available implementations of the bus are available. The three protocols described in [3] are:

- *Advanced High Performance Bus* (AHB) is a system backbone bus, intended for the connection of devices such as processors and on-chip memory caches.
- *Advanced System Bus* (ASB) is similar to AHB, but is not specifically targeted at high performance systems.
- *Advanced Peripheral Bus* (APB) is designed for low power peripherals; and has a correspondingly simpler functionality.

High level comparisons of these protocols can be found in [11]. In this paper we model AHB. This is because, unlike APB, it is intended for on-chip components as a system backbone, is therefore more fundamental to co-design systems; and is a more advanced protocol than ASB.

An AHB bus is essentially a central multiplexor and controller. Components connected to the bus request transfers and the bus arbitrates to whom, when, and under what conditions

the bus is granted. It is also responsible for multiplexing data, address, and control signals to the correct destinations. A typical AHB system contains the following components:

- *AHB master*: A master initiates read and write operations by providing address and control information. Only one master may actively use the bus at a time.
- *AHB slave*: A slave responds to a read or write operation within a given address-space. The slave signals back to the master the success, failure, or waiting of the transfer.
- *AHB arbiter*: The arbiter ensures only one master at a time initiates data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as *highest priority* or *fair access* can be implemented depending on application requirements.
- *AHB decoder*: The decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. It may be thought of as multiplexing shared lines of communication.

An AHB system consists of a collection of masters, slaves, a single arbiter, and a decoder managing accesses to the communication interconnect lines. A component which has a master interface may also have a slave interface.

A transaction starts with a master requesting the bus. When appropriate, the arbiter grants a master control of the bus. The master then drives control and address information and handshakes this with the destination slave, before driving the actual transaction data—which may be from the master to the slave (a write transaction) or from a slave to a master (a read transaction). The transaction completes either when the slave has transferred all of the data that the master required, or when the arbiter has called it to a halt for some overriding reason.

3.1. An example AHB system

Figure 1 shows the structure of an AHB system. The master and slave are identified by individual x tags—a more complex system would have more tagged lines unique to given masters and slaves. The diagram shows the various signals communicating between components. Where a line connects exactly two components (in this case because only one master and slave have been included) a simple arrow is used; where a signal is common to more than two components the lines fan out with a solid dot. Dashed lines are used in the diagram where lines cross solely to avoid confusion. The signals *hclk* and *hreset*, which are common to all components are listed in the box for each component. Arrows connecting components in this diagram are implemented as synchronizations in the CSP. Care must be taken with arrows parameterized with master and slave numbers though, as these are implemented as interleavings unique to each master as per the previous comments. The model of the bus can be seen to emerge from this diagram as a CSP process with an alphabet corresponding to the interface of the arbiter and decoder, controlling a B machine which captures the internal state.

3.2. Related work

There has been a lot of interest in understanding, and verifying the AMBA bus. Some of this interest is in the bus itself, and other interest is in general approaches to verifying System on

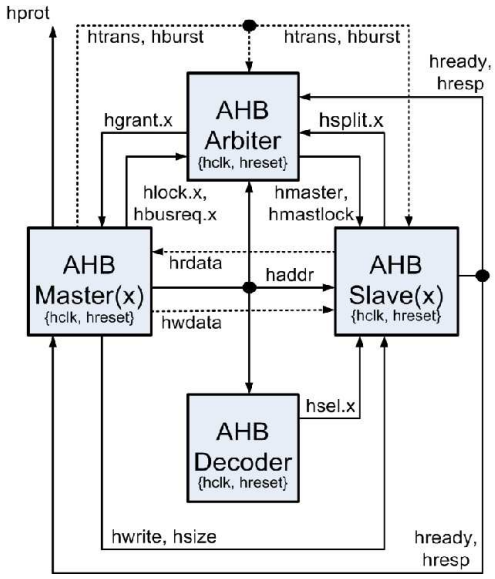


Figure 1. An example AHB system with one master and one slave

Chip architectures and implementations. For instance, [19] presents a platform incorporating the AMBA bus and ARM7 processor. Standard behavioural properties are then verified, and the resulting specification is used to develop specifications of other components for a given System on Chip platform. Verification uses a number of tools, but is centered around the HOL98 theorem prover. This work differs to ours in several respects. Firstly, it aims to provide a platform for an entire System on Chip application, whereas we are concerned specifically with the bus protocols. Secondly, there is no direct route to an implementation language, and we are interested in relating our specifications directly to *Handel-C* code. Thirdly, it uses a theorem prover to verify properties, and we are interested in automating our verification by using a model-checker.

Other works include [2]. This work also uses HOL for verification purposes. The goals are similar to ours in that it aims to build a formal model from the English language specification document, and investigate whether or not the English language specification is unambiguous and error-free. It has similar limitations in that it does not prove the system correct for arbitrary numbers of components, only for specific instances, and similar successes in that it models both control and data properties; however our work goes further in the attempt to build a model from which an implementation may be (automatically) derived.

Commercial tools are available. For instance SolidAHB [7] checks to ensure that components connected to the bus correctly adhere to the bus protocols. This aim is replicated in our work:

we wish to construct a model against which component specifications may be checked for compliance. The difference with our work is in the way the models are checked: SolidAHB tests a component against a set of test vectors, whereas our model-checking approach exhaustively tests the system for rogue states. When a rogue state is detected, a counter-example is given and the model must be amended to remove the possibility. Other versions of the tool include SolidPC, which is intended to check newer versions of the AHB protocol; and both Cadence and Synopsis incorporate testing and checking tools in their product suites.

4. A MODEL OF THE AHB ARBITER

In this section, we develop the model of the bus. The B machine is given in Section 4.1, and the CSP controller in Section 4.2. For each, the syntax used is as accepted by ProB. For the B, this is valid input to the B-Toolkit, and for CSP, this is valid input to FDR[9].

4.1. A B machine describing internal state

Definition 1. *Local variables (registers) and types*

```
SETS BurstType = { SINGLE, WRAP4 }

CONSTANTS MaxMaster
PROPERTIES MaxMaster = 3

DEFINITIONS  MASTER == 0..MaxMaster

VARIABLES XX,
          YY,
          ZZ,
          YYlatched,
          ZZlatched,
          Burst,
          Burstlatched,
          BurstCount,
          BurstCountlatched

INVARIANT XX           <: MASTER                          &
          YY           <: MASTER                          &
          ZZ           <: MASTER                          &
          YYlatched    <: MASTER                          &
          ZZlatched    <: MASTER                          &
          Burst         : BurstType           &
          Burstlatched  : BurstType           &
          BurstCount : 0..4                                    &
          BurstCountlatched : 0..4                             &
          ((BurstCountlatched > 1) => (Burst = Burstlatched))
```

Local state is modelled in terms of clocked, synchronous registers. That is, each register (or variable) has a value on a given clock cycle which does not alter on that clock cycle. If written

to on a clock cycle, it takes on the new value *only when the clock ticks*. If it is not written to, the value does not change on the next cycle. Every register updates simultaneously. The invariant given in Definition 1 contains the type declarations for each local piece of state; and a further conjunct that is used (in Section 5) for verification purposes.[†]

A master lodges a request by setting its request line high, and the arbiter chooses from all masters requesting the bus on a given cycle. A request is current *only when the request line is high*. Requests may be for either a locked, or an unlocked transaction. The set YY records all the masters that have set their request line on the current cycle. In this clocked synchronous model, this request is not stored in the arbiter until the clock ticks. *YYlatched* represents all those masters who lodged a request for the bus on the previous cycle, but did not ask for the bus to be locked. The set ZZ represents all those masters who lodged a request to lock the bus, and *ZZlatched* represents those who lodged a request to lock the bus on the previous cycle and also backed this up with a request for the bus. XX records which masters have not lodged a request on the current cycle allowing us to ensure that multiple requests cannot be made; while *Burst* and *BurstCount* relate to control for the current transaction.

Initially, no masters have lodged a request on the current cycle, and no masters could have lodged a request on the previous cycle. *YYlatched* is non-empty: this corresponds to a default master (0) always being assumed to have requested the bus. This feature captures the AMBA requirement that there should always be a default master to allocate the bus to—even when no requests have been made.

Definition 2. *Initialisation*

```
INITIALISATION XX          := MASTER       ||
               YY          := {}           ||
               ZZ          := {}           ||
               YYlatched   := {0}          ||
               ZZlatched   := {}           ||
               BurstCount  := 0            ||
               BurstCountlatched := 1      ||
               Burst       := SINGLE       ||
               Burstlatched := SINGLE
```

When a master requests the bus (*Request*), it is recorded by moving its index from XX to YY. It may request that a transaction is locked (*LockedRequest*): this is recorded by placing its index in ZZ. Variables recording requests on the previous cycle remain unchanged.

Definition 3. *Recording requests and locks for the bus*

```
Request(xx) =
PRE xx : MASTER THEN
  XX        := XX - {xx}  ||
  YY        := YY \/ {xx}
END;
```

[†]We restrict the transaction types to SINGLE and WRAP4 in this paper for two reasons: firstly to simplify the presentation and secondly to reduce the state space required in the model-checker.

```
LockedRequest(xx) =
PRE xx : MASTER THEN
  ZZ        := ZZ \/ {xx}
END;
```

YYlatched records all masters on the previous clock cycle who requested an *unlocked* transaction, and *ZZlatched* records all of those requesting a *locked* transaction. When the arbiter chooses which master is to be granted the bus on the next cycle, it non-deterministically selects an element from the union of these two sets.

Definition 4. *Choosing a master to which to grant the bus*

```
xx <-- Choose =
BEGIN
  xx :: YYlatched \/ ZZlatched
END;
```

A feature of the CSP‖B approach that is maintained in ProB is the requirement that state in the B machine is not accessed directly—a query operation should be provided in the machine instead. *GetBurstCount* is one such operation, and returns the number of ticks the current transaction has remaining.

Definition 5. *Testing for burst sizes*

```
xx <-- GetBurstCount =
BEGIN
  xx := BurstCountlatched
END;
```

When the type of burst is specified, a fixed length is assumed, and recorded by the operation *SetBurst*. A variable length transaction—given by the type *SINGLE*—is assumed to be a fixed length of one burst, and the controlling master is responsible for retaining the bus by continually re-asserting the request.

Definition 6. *Setting the burst type*

```
SetBurst(xx) =
PRE xx : BurstType THEN
  Burst := xx ||
    IF xx = WRAP4  THEN BurstCount := 4
    ELSIF xx = SINGLE THEN BurstCount := 1
  END
END;

SetTrans(xx) = PRE xx : BurstType THEN skip END;
```

The operation *tock* is carried out exactly when the clock ticks, and implements the clocked synchronous behaviour. When the clock ticks, a new cycle begins. No masters may have requested the bus yet on this new cycle, so XX is maximal, and YY and ZZ are emptied.

YYlatched takes on the value that *YY* held, ignoring all those who had also set the lock line high. It therefore holds all of those masters who requested an unlocked transaction on the clock cycle just ending. *ZZlatched* takes on all those masters who set the lock line high *and* requested the bus: the effect being that if a master erroneously set the lock line high but did not request the bus, it will be ignored. In case there were no requests lodged, it is assumed that the default master (0) must have lodged a request for an unlocked transaction. Finally, the type of the bus on the current cycle is stored, along with a note about any new burst type that may have been input.

Definition 7. *Synchronous clocked updates*

```
tock =
BEGIN
  XX        := MASTER                                          ||
  YY        := {}                                              ||
  ZZ        := {}                                              ||
  IF YY={} THEN YYlatched:= {0} ELSE YYlatched := YY - ZZ END ||
  ZZlatched := YY /\ ZZ                                        ||
  IF BurstCount > 0 THEN  BurstCount := BurstCount - 1 END     ||
  BurstCountlatched := BurstCount                              ||
  Burstlatched := Burst
END;
```

4.2. The CSP controller

The process *LOCKED_TRANS* listens on the request lines. When one goes high (*req*) it calls a B operation that records this. The same is true of the lock line *hlock*. When the current burst is in its last cycle ($BC == 1$) the highest priority master is chosen (*Choose*) and indicated (*hgrant*); and when the current burst has finished, the incoming burst indicates its type (*hburst* and *htrans*). The arbiter indicates it is ready for the incoming burst to begin (*hready*). When the current burst is in any cycle other than the final tocks, these signals are ignored. The model monitors, but does not interfere with the data and address lines. [‡]

Definition 8.

```
LOCKED_TRANS =
  GetBurstCount?BC -> (
    BC==1 & Choose?Highest -> hgrant!Highest -> LOCKED_TRANS'(BC)
    []
    BC != 1 & LOCKED_TRANS'(BC) )

LOCKED_TRANS'(BC) =
    tock ->
      GetBurstCount?BC -> (
        BC==1 & Choose?Highest -> hgrant!Highest -> LOCKED_TRANS'(BC)
        []
```

[‡]We use the event *tock* to denote a clock tick as "tick" '(✓) is commonly used in CSP to denote termination.

```
       BC != 1 & LOCKED_TRANS'(BC) )
    []
    ([] m : Masters @ req.m -> Request!m -> LOCKED_TRANS'(BC))
    []
    ([] m : Masters @ hlock.m -> LockedRequest!m -> LOCKED_TRANS'(BC))
    []
    BC==0 & ( htrans?xx -> SetTrans!xx -> LOCKED_TRANS'(BC)
              []
              hburst?xx -> SetBurst!xx -> LOCKED_TRANS'(BC)
              []
              hready -> LOCKED_TRANS'(BC) )
    []
    BC>=1 & ( htrans?xx-> LOCKED_TRANS'(BC)
              []
              hburst?xx -> LOCKED_TRANS'(BC) )
    []
    ([] x : {hmaster,hmastlock,hrdata,haddr} @ x -> LOCKED_TRANS'(BC))
```

Definition 8 does not constrain how many times on each clock cycle an event may occur, but the B machine assumes a master may only record one request per cycle. This constraint is captured in the CSP by placing Definition 8 in parallel with processes describing this constraint (Definition 9—although we omit the definitions of some constraints as they may be easily reconstructed). This process insists that when a request is lodged, the clock must tick before it may be lodged again; however the clock may tick an indeterminate number of times without a request being lodged. Other constraints are that *hready* may go high at most once per cycle, and that the arbiter must choose and grant the highest priority master on each cycle.

Writing the behavioural constraints in separate parallel processes in this way is a stylistic choice: they could have been added in a more implicit manner. However, in adopting this style the behavioural constraints are up-front: readily identifiable and easily changed should the model require adaptation or further development.

Definition 9. *Constraining requests*

```
REG_REQ(x) = req.x -> tock -> REG_REQ(x) [] tock -> REG_REQ(x)
REG_REQS = [|{| tock |}|] x : Masters @ REG_REQ(x)

REG_CHOOSE =
  Choose?Highest ->
    hgrant!Highest ->
      tock ->
        (hburst?any -> SKIP ||| htrans?any -> SKIP);
          tock -> REG_CHOOSE
  []
  tock -> REG_CHOOSE
  []
  hburst?any -> REG_CHOOSE
  []
  htrans?any -> REG_CHOOSE
```

```
REGULATE =
  ( ( REG_HREADY [|{| tock |}|] REG_CHOOSE )
    [|{| tock |}|]
    ( REG_REQS [|{| tock |}|] REG_LOCKS ) )
  [|{| tock |}|]
  ( (REG_HMASTLOCK[|{| tock |}|]REG_HMASTER)
    [|{| tock |}|]
   REG_TRANSFER )
```

The main process is simply the unconstrained model of locked transactions, running in parallel with the constraints that are appropriate for this model. In this paper, we omit unlocked requests to simplify the model.

Definition 10. *The main controller*

```
ORIG_MAIN =
       LOCKED_TRANS
       [|{| hready,hgrant,req,hlock, hmastlock,hmaster,haddr,
            hrdata, Choose,hgrant, hburst, htrans, tock |}|]
       REGULATE
```

5. ANALYSIS OF THE MODEL

In this section, we discuss some analysis that can be done on this model using ProB, and show how ProB can be used to check properties either of the B machine in isolation, or of the combination with CSP. [11] demonstrates the use of ProB in developing the model.

Although ProB supports animation, much of its power derives from its ability to perform model-checking, either on a stand-alone B machine, or on a CSP and B combination. Various properties can be checked through model-checking. The property we focus on here is invariant checking: that the machine can never reach a state in which its invariant is false. Properties are expressed as clauses in the invariant, and then investigated through the model-checker.

As an example, we have considered the property that the burst variable should not be reset while a burst is in progress. Recall that a burst value is set when a master obtains a lock on the bus. It will then have control of the bus, and will not release it, until the burst has completed. The value corresponding to the time remaining for the burst is tracked in the variable *BurstCount* within the B machine: this is set at the same time as *Burst*.

We wish to express this property as a requirement that *Burst* should not change while an existing burst is underway. To express this, we make use of the variables *Burstlatched* and *BurstCountlatched* which track the values of *Burst* and *BurstCount* from the previous clock cycle. The property is captured as the requirement that if the burst is not about to complete—i.e. is not on the last clock tick—then a new burst type should not be set: *Burst* should be the same as *Burstlatched*. Formally, this is given as $((BurstCountlatched > 1) \Rightarrow (Burst = Burstlatched))$ and incorporated into the invariant of the B machine.

Model-checking the stand-alone B machine with this assertion finds that the invariant is not always true. A trace given by ProB which leads to the violation of the invariant is

given in Example 1. This trace brings us to a state where $BurstCountlatched = 2$, and yet $Burst = INCR$ and $Burstlatched = SINGLE$ are different, indicating that *Burst* has just changed.

Example 1. *A counter-example produced by ProB*

$$\langle\ initialise\_machine, LockedRequest(2), LockedRequest(3), SetBurst(SINGLE), Request(2), tock, SetBurst(INCR)\rangle$$

□

This violation is expected: the AHB state machine in isolation will not ensure that the desired assertion is met. It is able to accept updates to the burst type at any stage, and this capability is what allows the invariant to be violated.

However, we expect the assertion to be true when the AHBstate machine is controlled by the CSP controller: the aim is that the controller ensures that updates to the burst type cannot occur in the middle of a burst. ProB is also able to model-check the AHBstate when it is under the control of the CSP controller. In this case it turns out that ProB does not find any invariant violations, confirming that the assertion is indeed valid for AHBstate in the appropriate context. This is what we had aimed for in the combined design.

5.1. Checking traces properties

Definition 11. *Observable good behaviours*

```
CheckAllocation =
  GetBurstCount.1 -> Choose?any -> hgrant!any -> tock -> CheckAllocation
  []
  ([] x : union({0},{2..16}) @ GetBurstCount.x -> tock -> CheckAllocation)

NEW_MAIN = MAIN [|{| GetBurstCount,Choose, hgrant, tock |}|] CheckAllocation
```

Other properties of the model are worth checking: in particular, properties of observable behaviours. For instance, we would expect that the bus will always allocate a new master on the penultimate cycle of a current transaction. A property such as this is not immediately expressible in terms of a state invariant in the B machine, but can be expressed as a predicate over the traces of the CSP controller. Definition 11 presents a process that contains only the good behaviour that we would expect: that an *hgrant* may be observed if, and only if, the current burst count is 1.

FDR can be used to check that the controller will only engage in these good behaviours by performing a refinement check such as that in Definition 12, which is shown to hold. The ability to check this offers a significant benefit over previous versions of ProB, which were not CSP_M compatible.

Definition 12. *A refinement check*

```
MostEvents = {| hready,hmaster,hmastlock,hrdata,haddr,req,Request,hlock,
                LockedRequest,hburst,SetBurst,htrans,SetTrans|}

assert CheckAllocation [T= ORIG_MAIN \ MostEvents
```

However, ProB can also check a simple traces property such as this. Using temporal logic, one can check to see when an operation is enabled in the B machine. However, here, we are concernced with an event in the CSP controller. We therefore create a dummy operation in the B machine, given in Definition 13. As the B machine and the CSP controller syncronise on common events, this operation will only be enabled when the CSP controller is not refusing it.

Definition 13. *A dummy operation*

```
hgrant(mm) = PRE mm : MASTER THEN skip END
```

Definition 14. *A temporal logic formula*

```
G( e(hgrant) => {BurstCountlatched=1})
```

The CSP traces specification stated that the bus would only be re-allocated when in the penultimate cycle of a transaction—that it can only occur between returning the result that $BurstCountlatched == 1$ and the clock next ticking. The formula given in Definition 14 states that if the B operation *hgrant* is enabled, then the value of *BurstCountlatched* must be 1. The relationship between the CSP controller and the B machine ensures that the model will refuse *hgrant* in any state where either the controller or the B machine refuse it. As it is always enabled in the B machine, a refusal can only come from the CSP. This LTL formula therefore captures the same predicate on the model as that expressed in the FDR refinement check: that the model will only ever not be refusing *hgrant* when $BurstCountlatched == 1$. This formula can be checked in ProB, and is found to be true of the combination of CSP controller and B machine.

6. COMPARISON WITH PREVIOUS RESULTS

In [11], an AHB model was described with an invariant similar to that given in this paper. However, our recent work has shown that the invariant is not true of the evolution of the model presented in this paper. When the search space checked by ProB was expanded to several million states, a counter-example was produced that proved the controller could permit the transaction type to change while the bus was engaged in a current transaction. If the proposed invariant had held, this would not have been the case.

Several lessons can be learned from this. The most important concerns the non-exhaustive nature of model-checking in ProB. When launching a temporal model-check, the user is required to state the number of nodes that the model-checker should visit. A positive result only guarantees that the invariant held for those nodes visited. This is in contrast to a model-checker such as FDR, that performs an exhaustive search of all states. Currently, we know of no technique that allows us, in general, to calculate a threshold of states for a given model in ProB necessary to prove a given invariant holds. If anything other than an exhaustive search is to be carried out, a threshold, with a supporting argument, would be necessary to give confidence in the result. The only reliable current technique of which we are aware is to ensure

that the model-checker is given a limit of the number of states to visit that exceeds the number of states in the model—and this can only be achieved by trial and error.

In addition to this, the incompatibility with CSP_M did not allow us to investigate the behavioural (traces based) properties of the previous model using FDR. The ability to use FDR to reason about the CSP controller, in addition to using ProB to reason about state invariants in conjunction with the CSP controller, is a very powerful feature. For instance, we were previously unable to prove the traces property discussed in Section 5 using ProB alone.

7. CONCLUSION AND DISCUSSION

In this paper we presented a case study where we modelled an existing on-chip bus protocol using a combination of CSP and B, and performed some analysis of the model using ProB and FDR. A driving aim of the paper was to investigate how CSP‖B, and ProB, may be used in a typical co-design development and verification. An interesting aspect of this case study is that it models an existing implementation, with the aim of providing a platform for formal analysis against components with which it is to be used. Thus in places, the model follows closely the behaviour described in the specification document, rather than some more abstract mathematical model. This has both benefits and drawbacks. Benefits include an easier discussion about the correctness of the model relative to the rather informal specification; while drawbacks include the constraints that this places on the construction of the model.

The AMBA bus is commonly used in co-design systems. Components on the bus may be processors, memory, or bespoke components. In building a model of the bus interacting via a CSP interface with bus components, we have found the combination of CSP and B sufficient to model signals, communications, and registers. The model in this paper is restricted to clocked synchronous hardware; an item of future work is to investigate the combination of CSP and B for asynchronous co-design systems.

We have attempted to remain faithful to the AMBA specification in the construction of our model, but as yet have not cross-checked it with an implementation. In fact, we believe that in doing so, we will discover behaviours that need revision. An item of future work would be to check this model relative to an existing implementation that is trusted and believed to be correct. Another item of future work would be to develop a master (or slave) component using CSP‖B and ProB, verify the correctness with respect to our model, derive an implementation and connect it to an implementation of the AMBA bus. Although subsequent testing of this implementation would not guarantee the correctness of the model, it would provide enough feedback to guide its evolution.

Another aim of this paper was to investigate the usage of ProB in a modeling and development exercise such as this, and to build on our existing work. In summary the existence of tool support proved useful in the development and prototyping phase although there are limitations in what can be achieved; and care must be taken with the size of models put through the model-checker.

One of the most interesting results to come out of ProB usage concerns the verification techniques that may be used. ProB produces counter-examples when a machine invariant is violated. Using machine invariants to capture safety properties is well understood in (amongst

others) the B community; using invariants over traces to capture safety properties proved by refinement checking is well understood in the CSP community. In this paper however, we augmented the B machine with extra information, designed to capture extra interactions with the CSP, such that the machine invariant could capture safe states. An uncontrolled B machine was shown to violate the invariant, whilst the B machine in parallel with the CSP controller was shown to respect the safety invariant. Although this example was simple, the important detail is the technique for lifting information into B. Further understanding and evolution of this technique of capturing traces invariants as properties of the B machine is an important item that we leave for future work; however the ability to use FDR to reason about trace invariants of the controller offers significant promise here.

ACKNOWLEDGEMENT

The work presented in this paper builds on that in [11], which was originally carried out under the AWE sponsored "Future Technologies for Systems Design" project whilst Alistair McEwan was at the University of Surrey. It has benefited from discussions with Helen Treharne, Wilson Ifill, Edd Turner and Michael Leuschel.

REFERENCES

1. J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Hasan Amjad. Verification of AMBA Using a Combination of Model Checking and Theorem Proving. In *Proceedings of AVoCS 2005*, pages 45–61. Elsevier, 2006.
3. ARM. Introduction to the AMBA bus. Technical Report 0011A, ARM, 1999.
4. B-Core. *B-Toolkit*.
5. Clearsy. *Atelier-B*.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985.
7. Averant Inc. *SolidAHB*.
8. M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FM 2003: The 12th International FME Symposium*, pages 855–874, 2003.
9. Formal Systems (Europe) Ltd. *FDR2, www.fsel.com*.
10. Alistair A. McEwan. *Concurrent Program Development*. PhD thesis, University of Oxford, 2006.
11. Alistair A. McEwan and Steve Schneider. Modelling and Analysis of the AMBA bus in CSP and B. In A. A. McEwan, S. Schneider, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, pages 379–398. IOS Press, 2007.
12. C. C. Morgan. Of wp and CSP. In W.H.J. Feijen, A. J. M. van Gesteren, D. Gries, and J. Misra, editors, *Beauty is our Business: a birthday salute to Edsger J. Dijkstra*. Springer-Verlag, 1990.
13. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
14. S. A. Schneider. *The B-Method: an introduction*. Palgrave, 2001.
15. S.A. Schneider. *Concurrent and Real-time Systems: The CSP approach*. Wiley, 1999.
16. S.A. Schneider and H.E. Treharne. Communicating B machines. In *ZB2002*, volume LNCS 2272, pages 416–438, 2002.
17. Steve Schneider and Helen Treharne. CSP theorems for communicating B machines. *Formal Aspects of Computing*, 17(4):390–422, 2005.
18. Steve Schneider, Helen Treharne, and Neil Evans. Chunks: Component verification in CSP‖B. In *IFM 2005*, volume LNCS 3771, pages 89–108, 2005.
19. Kong Woei Susanto and Tom Melham. An AMBA-ARM7 Formal Verification Platform. In J. S. Dong and J. Woodcock, editors, *ICFEM*, pages 48–67. LNCS, Springer-Verlag, 2003.
20. H. E. Treharne. *Combining control executives and software specifications*. PhD thesis, Royal Holloway, University of London, 2000.