Modeling and analysis of the AMBA bus using CSP and B

Alistair A. Mcewan
University of Derby
47 PUBLICATIONS **128** CITATIONS
SEE PROFILE

Steve Schneider
University of Surrey
196 PUBLICATIONS **4,933** CITATIONS
SEE PROFILE

Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007

Modeling and analysis of the AMBA bus using CSP and B

Alistair A. McEwan and Steve Schneider [1],

University of Surrey, U.K.

Abstract. In this paper, we present a formal model and analysis of the AMBA Advanced High-performance Bus (AHB) on-chip 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. We demonstrate how the model is developed from the informal ARM specification of the bus. Analysis is performed using the model-checker ProB. The contribution of this paper may be summarised as follows: presentation of work in progress towards a formal model of the AMBA AHB protocol such that it may be used for inclusion in, and 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 the ProB tool in performing this analysis. The work in this paper was carried out under the Future Technologies for Systems Design Project at the University of Surrey, sponsored by AWE.

Keywords. CSP||B, AMBA, formal modeling, ProB, co-design

Introduction

In this paper we present a model of the AMBA Advanced High-performace Bus (AHB) in the formalism CSP||B, and investigate analysis of the 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. Our aim is to 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. 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. This work has been carried out within the AWE funded project 'Future Technologies for System Design' at the University of Surrey, which is concerned with formal approaches to co-design.

Overview

The paper begins in section 1 by presenting some background information on CSP||B, ProB, and the AMBA bus. including notes on the main AMBA protocols. This is followed in section 2 by a description of the protocol about which this paper is concerned. Section 3 presents the CSP||B/ProB model. A discussion about the types of analysis that can be done on this model is presented in section 4, and some conclusions are drawn in section 5.

The contribution of this paper can be summarised as follows: a demonstration of modeling components used in a typical co-design environment using CSP||B, an evaluation of

[1]Corresponding Author: *Alistair A. McEwan, Department of Computing, University of Surrey, Guildford, U.K. GU2 7XH.* E-mail: `a.mcewan@surrey.ac.uk`

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.

1. Background

1.1. Combining CSP and B

CSP||B [18,15] is a combination of the process algebra CSP [6,12,14] and the language of abstract machines supported by the B-Method [1,13]. A *controlled component* consists of a B machine in parallel with a CSP process which is considered as the controller. 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[16], and to combine them into larger systems[17]. The approach taken in this paper differs in that it applies a model-checker to the CSP||B in order to achieve verification.

B machines

The B-Method develops systems in terms of *machines*, which are components containing state and supporting operations on that state. They are described in a language called *Abstract Machine Notation*. The most important aspect of B to understand for this paper is that B 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 [1,13], and tool support is provided by [3,4].

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.

Example 1 *The format of a B operation*

$$oo \longleftarrow op(ii) = \text{PRE } P \text{ THEN } S \text{ END}$$

□

The format of a B operation is given in example 1. The declaration $oo \longleftarrow op(ii)$ introduces the operation: it has name op, a (possibly empty) output list of variables oo, and a (possibly empty) input list of variables ii. The precondition of the operation is predicate P. This must give the type of any input variables, and can also give conditions on when the operation can be called. If it is called outside its precondition then divergence results. Finally, the body of the operation is S. This is a *generalised substitution*, which can consist of one or more assignment statements (in parallel) to update the state or assign to the output variables. Conditional statements and nondeterministic choice statements are also permitted in the body of the operation. Other clauses are also allowed, for instance regarding machine parameters, sets and constants. For an example B machine, see section 3 where the B machine that is the subject of this paper is introduced.

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 accepted are the *Traces* model, and the *Failures/Divergences* model. Full details can be found in [12, 14]. 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 (such as a livelock loop) 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[8].

CSP semantics for B machines

Morgan's CSP-style semantics [11] 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 [11] this is $\neg wp(e_1;\ e_2;\ \dots;\ e_n, false)$, or in Abstract Machine Notation $\neg([e_1;\ e_2;\ \dots;\ e_n]false)$. (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 of the semantics of B machines appear in [18]. The CSP semantics for B machines enables the parallel combination of a B machine and a CSP process to be formally defined in terms of the CSP semantics.

The term *CSP controller* P means a process which has a given set of control channels (events) C. The controlled B machine will have exactly $\{|\ C\ |\}$ [1] as its alphabet: it can communicate only on channels in C where a channel name corresponds to an operation in the machine. To interact with the B machine, a CSP controller makes use of control channels which have both input and output, and provide the means for controllers to synchronise with

[1]The notation $\{||\}$ is used to fully qualify channel sets in CSP. For instance, assuming channel $X : Bool$, $\{|\ X\ |\}$ is the set $\{X.true, X.false\}$.

B machines. 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[15], 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 on the syntax of CSP controllers 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. This includes parallel and interleaving operators, as well as prefixing, sequential composition, recursion, and the various forms of choice.

ProB tool support

ProB [7] 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 B machine captures state, and the CSP characterises interactions with the environment, normally restricting the states in which a related B operation may be invoked. 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, although we remark where differences are significant. [2]

The version of CSP that is implemented in ProB bears a resemblance to, and draws some inspiration from, the CSP_M of FDR. Despite this, there are several differences to CSP_M. For instance, there are no replicated operators, channel type declarations are not supported, and there is no support for the functional language included in FDR. A reader familiar with CSP_M will easily comprehend the CSP supported by ProB, although will notice some of these differences. In this paper, we remark on the differences between the ProB CSP and CSP_M where differences are significant.

1.2. The AMBA bus

The *Advanced Microcontroller Bus Architecture)* (AMBA) is an on-chip communication standard for embedded micro controllers[2]. 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 [2] 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.

[2]The differences are in the theoretical basis of the combination, and a discussion is not within the scope of this paper. The interested reader is referred to [7] and [16,17].

	AHB	ASB	APB
High performance	✓[2],1.1,1.3	✓[2],1.3	
High clock rate	✓[2],1.1		
System backbone	✓[2],1.1	✓[2],1.1	
On-chip memories	✓[2],1.1	✓[2],1.1	
Off-chip memories	✓[2],1.1	✓[2],1.1	
External memory interfaces	✓[2],1.8	✓[2],1.8	
Low power optimised			✓[2],1.1,1.3
Used in conjunction with AHB			✓[2],1.1
Used in conjunction with ASB			✓[2],1.1
Pipelined operation	✓[2],1.3	✓[2],1.3	
Multiple bus masters	✓[2],1.3	✓[2],1.3	
Burst transfers	✓[2],1.3		
Split transactions	✓[2],1.3		
Latched address and control			✓[2],1.3
Simple interface			✓[2],1.3
Suitable for many peripherals			✓[2],1.3

Figure 1. High level description of properties of an AMBA bus

A fourth protocol, *AXB*, is also used in high performance systems but is not considered in this paper. Figure 1 presents comparisons of the three protocols described above.

2. Components in the AMBA AHB protocol

In this paper we model AHB. This is because, unlike APB, it is intended for on-chip components as a system backbone, and is therefore more fundamental to co-design systems; and it is a newer, 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 transfered all of the data that the master required, or when the arbiter has called it to a halt for some overriding reason.

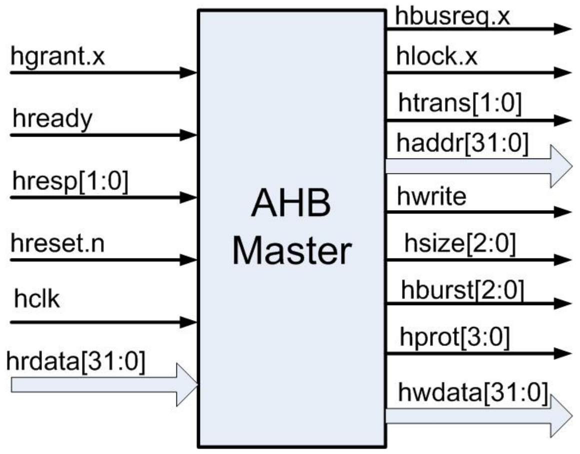


Figure 2. The AHB master interface diagram

In the following sections, we construct the interfaces of each component, allowing for a prototype construction that is readily checkable against the ARM specification.

2.1. The interface of an AHB master

In figure 2 an AHB master is shown in terms of inputs and outputs. An input is an arrow leading into the master, and an output is an arrow leading out of a master. The width in terms of bit indices is given. Where this is a single bit—therefore either high or low—no width is given.

A master requests the bus by setting its $hbusreq.x$ signal high (where x is a unique identifier); and may indicate that it does not wish its allocation to be interleaved with other transactions by also setting its $hlock.x$ signal. The transfer type is denoted by a range of signals on $htrans$, and the direction of the transfer by setting $hwrite$ either high or low. The size is given on $hsize$, the number of beats on $hburst$, and $hprot$ is used if there is further user level protection required. A master is told it is the highest priority waiting when $hgrant.x$ is high, and the bus is ready for use when $hready$ is high. Responses from the active slave are on $hresp$, and data can be read from a slave on $hrdata$. Each master has a clock pulse and reset line.

This is described in terms of sets of CSP channels in definition 1 for a given master x. The set of channels leading to all masters would be achieved by disregarding the identifier x for an individual master. This distinction between channels global to the masters, and channels individual to each master is important as it dictates synchronization sets and interleaving when processes are composed in the CSP model. [3]

Definition 1 *AHB Master x actuates and senses*

```
OUTPUTS(x) = {| hbusreq.x, hlock.x, htrans, haddr, hwrite, hsize,
                hburst, hprot, hwdata |}
INPUTS(x)   = {| hgrant.x, hready, hresp, hreset.x, hclk, hrdata  |}
```

□

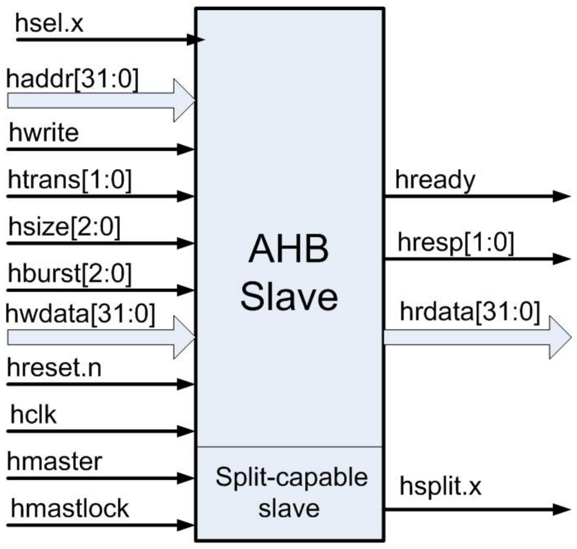


Figure 3. The AHB slave interface diagram

2.2. *The interface of an AHB slave*

When a slave has finished a current transaction, it sets $hready$ high. Other responses, such as error conditions, can be relayed back to the master on $hresp$. If the transaction is a read transaction, data is placed on the $hrdata$ line The $hsel.x$ is unique to a given slave x, and when high indicates the current transfer is intended for that slave. The signals $hwrite$, $htrans$, $hsize$ and $hburst$ are slave ends of master outputs mentioned previously. Each slave has a reset and clock line. This is described in terms of sets of CSP channels in definition 2. The signals $hmaster$, $hmastlock$, and $hsplit.x$ are concerned with split transactions and are not considered in our model, although we include them in the definition for completeness.

Definition 2 *AHB Slave x actuates and senses*

```
OUTPUTS(x) = {| hready, hresp, hrdata, hsplit.x |}
INPUTS(x)   = {| hset.x, haddr, hwrite, htrans, hsize, hburst,
                 hwdata, hreset.x, hclk, hmaster, hmastlock |}
```

□

2.3. *The interface of an AHB arbiter*

The arbiter ensures that only one master believes it has access to the bus at any one given time (and this may be a default master if necessary). It achieves this by monitoring request lines from masters wishing access, and selecting a master to grant the bus to from those requests. The description in [2] does not prescribe a resolution strategy; in this model we abstract using non-determinism. Figure 4 shows an AHB arbiter in terms of inputs and outputs. This is described in terms of sets of CSP channels in definition 3.

Definition 3 *AHB arbiter actuates and senses*

```
OUTPUTS = {| hgrant, hmaster, hmastlock |}
INPUTS  = {| hbusreq, hlock, haddr, hsplit, htrans,
             hburst, hresp, hready, hreset.x, hclk |}
```

□

[3]In considering channels, sets, and types, the first difference between ProB CSP and CSP_M appears. CSP_M requires channels to be typed. For instance, the single-bit channel $hwrite$ could be declared $chan\ hwrite : 0 \mid 1$; however ProB does not support typing, and instead infers types from values being passed.

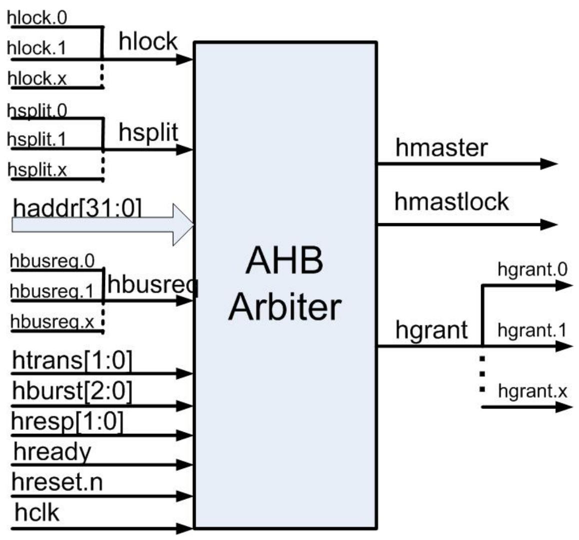


Figure 4. The AHB arbiter interface diagram

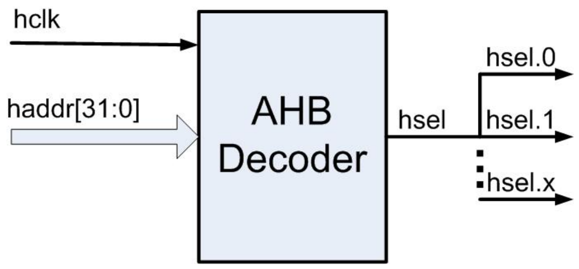


Figure 5. The AHB decoder interface diagram

2.4. The interface of an AHB decoder

The decoder acts as a multiplexor, and decodes data on the address bus identifying the slaves that transactions are intended for; and setting the relevant slave select line high. Figure 5 shows a decoder in terms of inputs and outputs, with the CSP channels in definition 4.

Definition 4 *AHB Decoder actuates and senses*

```
OUTPUTS = {| hsel |}
INPUTS   = {| haddr, hclk |}
```

□

2.5. An example AHB network

Figure 6 shows an example AHB network, comprising a master, slave, arbiter, and decoder. The master and slave are identified by their 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. For further clarity in the diagram, 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

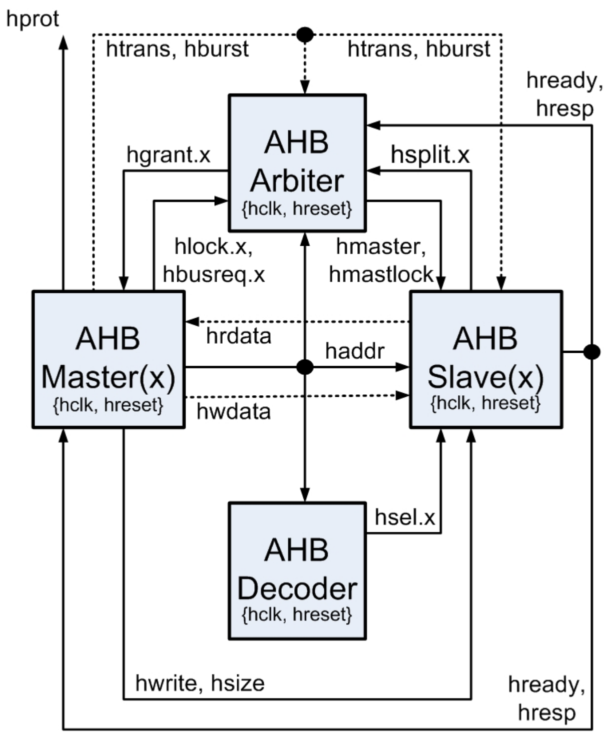


Figure 6. An example AHB system with one master and one slave

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. A model of the AHB components

In this section, we develop the model of the bus. The B machine is given in section 3.1, and the CSP controller in section 3.2. For each, the syntax used is as accepted by ProB. For the B, this is valid input to the B-Toolkit.

3.1. A B machine describing internal state

Local state is modeled 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 5 contains the type declarations for each local piece of state; and a further conjunct that is used (and described later in section 4.2) for verification purposes. [4]

Definition 5 *Local variables (registers) and types*

```
SETS BurstType = { SINGLE, INCR, WRAP4,INCR4, WRAP8,INCR8, WRAP16,INCR16 }
VARIABLES XX, YY, ZZ, YYlatched, ZZlatched,
          Burst, Burstlatched, BurstCount, BurstCountlatched

INVARIANT
  XX <: 0..15          & YY <: 0..15          & ZZ <: 0..15 &
```

[4]For a full reference of the syntax of a B machine and notation, the reader is referred to [13].

```
  YYlatched <: 0..15  & ZZlatched <: 0..15  &

  Burst       : BurstType &
  Burstlatched : BurstType &
  BurstCount : 0..17     &
  BurstCountlatched : 0..17 &
  ((BurstCountlatched > 0) => (Burst = Burstlatched))
```

□

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. If a master does not have its request line high it is assumed not to want the bus—i.e. a request is current *only when the request line is high.* Requests may be for either a locked, or an unlocked transaction. 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. $YYlatched$ contains all of the masters that lodged a request on the previous cycle: it is the value of YY on the previous cycle—the successful master will be drawn from the set $YYlatched$. The same clock synchronous behaviour is true of ZZ and $ZZlatched$, used to record requests for locked transactions. XX records which masters have not lodged a request on the current cycle (ensuring that each master may only lodge one request per cycle); 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. Curiously though, $YYlatched$ is non-empty: this corresponds to a default master (0) always being assumed to have requested the bus.[5]

Definition 6 *Initialisation*

```
INITIALISATION XX                := 0..15  ||
               YY                := {}     ||
               ZZ                := {}     ||
               YYlatched         := {0}    ||
               ZZlatched         := {}     ||
               Burst             := INCR   ||
               Burstlatched      := INCR   ||
               BurstCount        := 0      ||
               BurstCountlatched := 0
```

□

When a master requests the bus, it is recorded by removing its index from XX and placing it in YY. No assumption is made about whether or not this was a locked request or not. The variables recording requests on the previous cycle remain unchanged.

Definition 7 *Recording a master's request for the bus*

```
  Request(xx) =
  PRE xx : 0..15 THEN
    XX        := XX - {xx}  ||
    YY        := YY \/ {xx}
  END;
```

□

A master may request that a transaction is locked, and this is recorded by placing its index in ZZ. No assumption is made about whether or not this master has actually requested a transaction, and requests recorded from the previous clock cycle remain unchanged.

[5]In B, || denotes that the assignments all happen concurrently.

Definition 8 *Recording a master's request to lock the transaction*

```
LockedRequest(xx) =
PRE xx : 0..15 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 9 *Choosing a master to which to grant the bus*

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

□

One artefact of ProB is an inability to directly access state in the B machine from CSP—an operation must be invoked with a return value. $TestLock$ returns a boolean value indicating if master identifier passed to it is currently granted rights to a locked transaction. $GetBurstType$ performs a similar function to test the burst type of a transaction, and $GetBurstCount$ indicates whether or not we are on the last element of a given burst. [6]

Definition 10 *Testing for locked transactions, bust types, and burst sizes*

```
xx <-- TestLock(yy) =
PRE yy : 0..15 THEN
  IF yy : ZZlatched THEN xx := TRUE ELSE xx := FALSE END
END;

xx <-- GetBurstType =
BEGIN
  IF Burst=SINGLE THEN xx:= TRUE ELSE xx:= FALSE END
END;

xx <-- GetBurstCount =
BEGIN
  IF BurstCount > 0 THEN xx := TRUE ELSE xx := FALSE END
END;
```

□

When the type of burst is specified, a fixed length is assumed, and recorded by the operation $SetBurst$. These lengths are one larger than might be expected due to the synchronous nature of the clocked assignments: they only really take on a meaningful value on the next clock cycle, by which time they will have been decremented by 1. 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.

[6]These values are boolean rather than $\mathbb{N}$ as ProB has not fully implemented CSP guards using tests on natural numbers—so this functionality must be moved into the B machine.

Definition 11 *Setting the burst type*

```
SetBurst(xx) =
PRE xx : BurstType THEN
  Burst := xx ||
  IF xx = INCR16 or xx = WRAP16 THEN BurstCount := 17 ELSE
  IF xx = INCR8  or xx = WRAP8  THEN BurstCount := 9  ELSE
  IF xx = INCR4  or xx = WRAP4  THEN BurstCount := 5  ELSE
  IF xx = SINGLE or xx = INCR   THEN BurstCount := 2  END END END END
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. This information is used for verification purposes in section 4.2.

Definition 12 *Synchronous clocked updates*

```
tock =
BEGIN
  XX        := 0..15                                               ||
  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
```

□

3.2. The CSP controller

In CSP||B and ProB, CSP controls when, and under what conditions, B operations can be invoked. An invocation of a B operation corresponds to the CSP controller engaging in an event of the same name, and parameters to the operation, and results of the operation, are passed in the types of the event.

The process $COLLECT\_REQUESTS$ listens on the request lines.[7] When one goes high (indicated by the process engaging in a $hbusreq$ event) it calls an operation in the B machine that records this. A lock line ($hlock$) may also go high, and when it does, another B operation is called. The $hready$ signal may also go high, indicating that the *current* transaction is ending. The $hgrant$ signal is used by the arbiter to indicate the highest priority request on the *previous* clock cycle, and this is achieved by calling the B operation $Choose$ that returns the value to the CSP. Finally, the clock may tick—indicated by the event $tock$.[8]

[7]Where behaviour is the same for multiple masters, we only include the CSP for master 0 to conserve space. This is because the CSP in ProB does not implement replicated operators, as a user of FDR may expect.

[8]We use the event $tock$ to denote a clock tick as “tick” ‘(✓) is commonly used in CSP to denote termination.

Definition 13 *Collecting requests*

```
COLLECT_REQUESTS =
  hbusreq.0 -> Request!0 -> COLLECT_REQUESTS
  [] hlock.0 -> LockedRequest!0 -> COLLECT_REQUESTS
  [] hready -> COLLECT_REQUESTS
  [] Choose.HighPri -> hgrant.HighPri -> COLLECT_REQUESTS
  [] tock -> COLLECT_REQUESTS
;;
```

□

Definition 13 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.[9] This constraint is captured in the CSP by placing definition 13 in parallel with processes describing this constraint. 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 14 *Constraining requests*

```
REG_HREADY = hready -> tock -> REG_HREADY [] tock -> REG_HREADY;;
REG_CHOOSE = Choose.HighPri -> hgrant.HighPri -> tock -> REG_CHOOSE;;
REG_REQ0 = hbusreq.0 -> tock -> REG_REQ0 [] tock -> REG_REQ0;;
REG_LOCK0 = hlock.0 -> tock -> REG_LOCK0 [] tock -> REG_LOCK0 ;;

REGULATE =
  ( REG_HREADY [|{tock}|] REG_CHOOSE )
  [|{tock}|]
  ( REG_REQS [|{tock}|] REG_LOCKS )
;;
```

□

Definition 15 presents the CSP process controlling locked transactions. A new transaction begins when the previous transaction ends with an *hready* signal. There are two possibilities here, corresponding to the first external choice in this process: that the arbiter may receive the *hready* signal before issuing an *hgrant* signal on a given clock cycle, or vica-versa. Subsequent behaviour is dependent upon whether or not the B machine indicates it is a locked transaction. At this point the clock ticks and the controller evolves into the transaction phase.

Definition 15 *Locked transactions*

```
LOCKED_TRANS =
  hready -> (
    hgrant.0 ->
      TestLock!0?RR ->
        tock -> (if RR then LOCKED_CTRL_INFO(0) else LOCKED_TRANS) )
  []
  hgrant.0 -> (
    hready ->
```

[9]The piece of syntax ;; indicates the end of a process definition.

```
      TestLock!0?RR ->
        tock-> (if RR then LOCKED_CTRL_INFO(0) else LOCKED_TRANS)
    []
    tock -> LOCKED_TRANS )
;;
```

□

In the control phase, the arbiter ensures the master locks the bus using $hmastlock$, and asserts control with $hmaster$. The master then dictates the burst type for the transfer—either fixed or variable length. Behaviour branches after the clock has ticked depending upon transfer type.

Definition 16 *Control phase of a locked transaction*

```
LOCKED_CTRL_INFO(PP) =
  hgrant?ANY ->
    hmastlock ->
      hmaster!PP ->
        hburst?TT ->
          SetBurst.TT ->
            GetBurstType?UU ->
              tock -> (if UU then LOCKED_VAR(PP) else LOCKED_FIXED(PP))
;;
```

□

In a locked transaction, the master is required to continually assert the lock lines while the transaction is in progress. The arbiter is required to assert the master that will be granted the bus on the next cycle *if the current transaction completes*. If the burst count is zero after the clock has ticked, then behaviour returns to monitoring for the next transaction, otherwise the current transaction continues to control the bus for another cycle.

Definition 17 *Control phase of a locked transaction*

```
LOCKED_FIXED_DATA(PP) =
  hmastlock ->
    hmaster!PP ->
      hburst?TT ->
        hgrant?ANY ->
          tock -> LOCKED_FIXED(PP)
;;

LOCKED_FIXED(PP) =
  GetBurstCount?XX ->
    ( if XX then LOCKED_FIXED_DATA(PP) else LOCKED_TRANS )
```

□

The main process is the process responsible for collecting requests, in parallel with the constraints placed upon it, in parallel with the process that marshals locked requests, and therefore implements the arbiter of figure 4 as well as implicitly implementing the multiplexing of control lines performed by the decoder. In this paper, we omit unlocked requests to simplify the model. The locked transaction marshaled synchronises with the request collector on the $hgrant$ and $hready$ signals—which is sufficient (in conjunction with the state stored in the B machine) for it to spot wheen the entire system is in a state corresponding to a locked transaction. All processes synchronise on the global clock event $tock$—which also causes the clocked synchronous behaviour in the B machine.

Definition 18 *The main controller*

```
MAIN =
  ( COLLECT_REQUESTS
    [|{ tock, Choose, hgrant, hready, hbusreq, hlock }|]
    REGULATE )
  [|{ tock, hgrant, hready }|]
  LOCKED_TRANS
;;
```

□

4. ProB 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. We also demonstrate the usefulness of ProB in developing the model because of the way it can be used to animate models. We document some experiences of the tool—some of which are mentioned above in the CSP model. We also discuss some contrasts with how FDR may be used in the development of a model—for instance, how FDR was used in the development of a CSP||B based Content Addressable Memory using *Circus* in [10].

4.1. Animating models using ProB

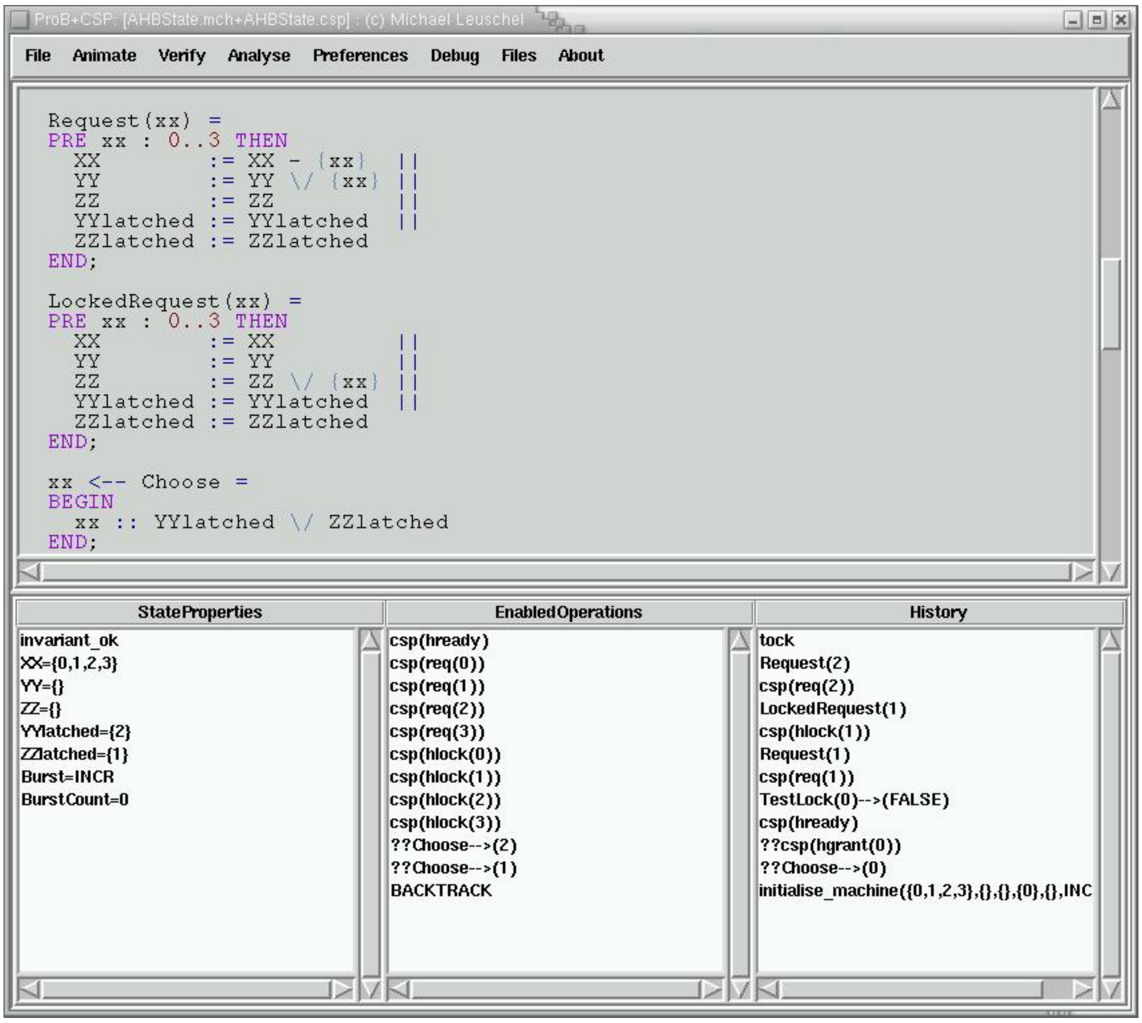
The initial use of ProB is in the construction of the CSP||B model, and particularly in the combination of the CSP controller and the underlying B machine. Animation in ProB allows the user to step through the behaviour of the CSP controller, at each step being offered a set of possible next steps to perform. The B machine is updated in the light of operation calls, and the updated state is exhibited.

This ability to step through the behaviour of the combined system supports exploration of its description, and enables immediate feedback on whether it exhibits the expected behaviour. Thus ProB is effective in supporting the construction of the formal model at the point it is being developed, and in ensuring consistency between the CSP and the B.

Figure 7 presents a snapshot of ProB animating the model. The uppermost window is an editor for the B machine. The bottom left window shows the state of each variable in the B machine, and a check on whether or not the state meets the machine's invariant. The bottom center window shows the CSP events currently on offer by the CSP controller (which includes available B operations). The bottom right window shows (from bottom to top) the trace of the animation so far.

Firstly, the machine is initialised with the $initialise\_machine$ call. As this begins the first clock cycle the default master 0 has been chosen by engaging in the B operation $Choose \rightarrow (0)$ and granted with the CSP event $hgrant(0)$. The $hready$ signal has occurred, indicating that new transaction may begin from this point in the trace onwards. As this is simply a default transaction, it is found to be unlocked by the B operation $TestLock(0) \rightarrow (FALSE)$. At this point, the arbiter starts receiving requests from masters wishing to use the bus on the next cycle. Master 1 lodges a request by synchronising with the controller on the event $req(1)$ and the Arbiter records this fact with the B operation $Request(1)$. The master confirms this is a locked transaction request with the event and operation $hlock(1)$ and $LockedRequest(1)$ respectively. Master 2 also lodges a request using the event and operation $req(2)$ and $Request(2)$ respectively. At this point, the clock ticks, updating all the synchronous registers.

The value of the B machine variables in the left window reflects the state of the B machine at this point. The current state satisfies the machine invariants. No masters have yet


ProB+CSP: [AHBState.mch+AHBState.csp] : (c) Michael Leuschel
File
Animate
Verify
Analyse
Preferences
Debug
Files
About

```
Request(xx) =
PRE xx : 0..3 THEN
  XX        := XX - {xx}  ||
  YY        := YY \/ {xx} ||
  ZZ        := ZZ         ||
  YYlatched := YYlatched  ||
  ZZlatched := ZZlatched
END;

LockedRequest(xx) =
PRE xx : 0..3 THEN
  XX        := XX         ||
  YY        := YY         ||
  ZZ        := ZZ \/ {xx} ||
  YYlatched := YYlatched  ||
  ZZlatched := ZZlatched
END;

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

StateProperties	EnabledOperations	History
invariant_ok	csp(hready)	tock
XX={0,1,2,3}	csp(req(0))	Request(2)
YY={}	csp(req(1))	csp(req(2))
ZZ={}	csp(req(2))	LockedRequest(1)
YYlatched={2}	csp(req(3))	csp(hlock(1))
ZZlatched={1}	csp(hlock(0))	Request(1)
Burst=INCR	csp(hlock(1))	csp(req(1))
BurstCount=0	csp(hlock(2))	TestLock(0)-->(FALSE)
	csp(hlock(3))	csp(hready)
	??Choose-->(2)	??csp(hgrant(0))
	??Choose-->(1)	??Choose-->(0)
	BACKTRACK	initialise_machine({0,1,2,3},{},{},{0},{},INC

Figure 7. A snapshot of animating the model (with 4 masters) in ProB

requested the bus on this current (i.e, second) cycle, so XX is maximal and YY and ZZ are empty. On the previous (first) cycle, master 2 requested an unlocked transaction and master 1 requested a locked transaction and this is reflected in the values of $YYlatched$ and $ZZlatched$. There is no transaction in progress, so the burst types hold their default value.

Finally we can see the events that can be performed in the current state (including B operations). Each of the masters may request the bus, and request that the transaction is locked by engaging in their respective CSP events req and $hlock$. The controller may invoke the operation to choose a master for the next clock cycle, and this operation may return the value 1, or 2. Notification of a current transaction ending may also be received. The $BACKTRACK$ alternative is for stepping backwards through the animation.

4.2. Model-checking using ProB

Although ProB supports animation, much of its power derives from its ability to perform model-checking, either on a stand-alone B machine, or else on a CSP and B combination. Various properties can be checked through model-checking. The property that we have focused on in this analysis is invariant checking: that the machine can never reach a state in which its invariant is false. Properties of interest can be 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 then captured as the requirement that if the burst had not finished on the previous clock cycle then a new burst should not be set: $Burst$ should be the same as $Burstlatched$. Formally, this is given as the statement $((BurstCountlatched > 0) \Rightarrow (Burst = Burstlatched))$ 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 2. 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. In fact, the same invariant violation can be reached through a shorter sequence of events, given in example 3.

Example 2 *A counter-example produced by ProB*

$$\begin{array}{l}\langle\, initialise\_machine, LockedRequest(2), LockedRequest(3),\\ \;SetBurst(SINGLE), Request(2), tock, SetBurst(INCR)\rangle\end{array}$$

□

Example 3 *A shorter counter-example*

$$\langle initialise\_machine, SetBurst(SINGLE), tock, SetBurst(INCR)\rangle$$

□

This violation is not unexpected: 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.

4.3. Experiences of CSP and B in ProB

This case study has exposed a number of experiences with using ProB. In this section we discuss some of these experiences. This discussion is intended to provide the reader with a guide as to practical, and mature, use of ProB in a typical CSP||B development, and why, and where, it may be of use.

- *Differences with CSP_M*: a number of differences with the CSP_M supported by FDR exist. Some of these are minor, and some more major. For instance, the syntax of the two is subtly different, some constructs in FDR are not supported by ProB, and the functional language in FDR is not supported in ProB. The impact is that a CSP script supported by FDR will not currently be directly supported by ProB and vice-versa. This is unfortunate as there is a wealth of experience and knowledge in using FDR that may not be directly applicable to a ProB script.

- *Structured development*: ProB does not have support for a structured development of a system of B machines, unlike tool support for the B-Method such as the B-Toolkit [3] and Atelier-B [4]. Although the B supported by the two is the same, ProB does not allow, for instance, included B machines in a script—there is only support for one machine per model. Other B machines must be manually included in-line. This is unfortunate as a project in the B-Toolkit requires manual intervention before being loaded into ProB; and this type of intervention should typically be avoided in high assurance systems.
- *Differences with CSP||B*: a characteristic feature of CSP||B is that a call to a B operation from a CSP controller can be hidden from external observations, with the result that only observations of the controller are possible. However, ProB handles hiding of controller calls to B operations differently. In hiding a call, the *call itself* becomes non-deterministic—there is no control over the value of parameters. This is in contrast to the CSP||B approach. This is unfortunate because it is an important semantic difference between ProB and CSP||B—although in a development/test cycle such as the one in this paper it is of minimal impact.
- *Animation*: the ability to animate models is very useful. This can be done for B machines using the B-Toolkit, or CSP processes using *Probe*[9]; but to be able to animate the combination of the two together means that many errors and inconsistencies can be caught early in the development cycle.
- *Model-checking vs theorem proving*: model-checking invariants in the B is useful. Model-checking is generally considered a more convenient route to verification than theorem proving because of its automatic nature. The B-Toolkit provides a theorem prover; to complement this with a model-checker is extremely valuable for developmental cycles as typically one would like to relieve the proof burden as much as possible.
- *Invariants over CSP processes*: a speculative usage of ProB that we have begun to explore through this case study is the use of invariants over CSP traces (or even failures) rather than just invariants in the B. To a user of FDR, the construction and asserting of a traces refinement in a specification is a useful tool in checking safety requirements[10]. A mechanism for specifying an invariant over traces of a process in ProB would, we expect, be a valuable addition to the tool; although we have not considered the theory about how such an addition could be formulated.

5. Conclusions and discussion

In this paper we have presented a case study where we modeled an existing on-chip bus protocol using a combination of CSP and B, and performed some analysis of the model using ProB. A driving aim of the paper was to investigate how CSP||B, and ProB, may be used in a typical co-design development.

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. The model in this paper therefore represents work in progress. An 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. The conclusions drawn from this are listed in section 4.3; in summary the existence of tool support for proved useful in the development and prototyping phase although there were limitations in what could be achieved and in the compatibility with tools for both CSP and B. A discussion of issues such as these—semantic and syntactic integration of formalisms and impact on associated tool support is held in [10,5].

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, as in section 4.2. 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.

Acknowledgements

This work was funded by AWE under the ‘Future Tchnologies for System Design’ project, and has benefitted from discussions with Wilson Ifill, Neil Evans, and Helen Treharne.

References

[1] J-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
[2] ARM. Introduction to the amba bus. Technical Report 0011A, ARM, 1999.
[3] B-Core. *B-Toolkit.*
[4] Clearsy. *Atelier-B.*
[5] C. Fischer. How to combine Z with a process algebra. *LNCS*, 1493, 1998.
[6] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985.
[7] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FM 2003: The 12th International FME Symposium*, pages 855–874, 2003.
[8] Formal Systems (Europe) Ltd. FDR: User manual and tutorial, version 2.82. Technical report, Formal Systems (Europe) Ltd., 2005.
[9] Formal Systems (Europe) Ltd. Probe user manual. Technical report, Formal Systems (Europe) Ltd., 2005.
[10] Alistair A. McEwan. *Concurrent Program Development.* DPhil thesis, The University of Oxford, 2006.
[11] 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.
[12] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[13] S. A. Schneider. *The B-Method: an introduction*. Palgrave, 2001.

[14] S.A. Schneider. *Concurrent and Real-time Systems: The CSP approach*. Wiley, 1999.

[15] S.A. Schneider and H.E. Treharne. Communicating B machines. In *ZB2002*, volume LNCS 2272, pages 416–438, 2002.

[16] Steve Schneider and Helen Treharne. CSP theorems for communicating B machines. *Formal Aspects of Computing*, 17, 2005.

[17] Steve Schneider, Helen Treharne, and Neil Evans. Chunks: Component verification in CSP||B. In *IFM 2005*, volume LNCS 3771, pages 89–108, 2005.

[18] H. E. Treharne. *Combining control executives and software specifications*. PhD thesis, Royal Holloway, University of London, 2000.