Validation of a Parameterized Bus Architecture Model

Julien Schmaltz and Dominique Borrione
TIMA-VDS, Grenoble, France

July 3, 2003

Abstract

In this paper, we present an experiment in the modeling of the AMBA-AHB virtual component and the proof of essential properties to validate the model. We prove the correctness of communications for an arbitrary number of masters and slaves.

1 Introduction

The design of *systems on chip* (SoC) results in the integration of pre-designed blocks, called *Virtual Components* (VC). In such a design flow, building the interconnection becomes the critical step, especially for the verification. This new paradigm introduces two challenges for formal methods: proving VCs and proving their composition. The bus, considered a VC [VSI01] is a *parameterized* structure, *i.e.* it is an *unbounded* system. In this paper, we present an experiment in the modeling of the AMBA-AHB virtual component, and the proof of essential properties to validate the model.

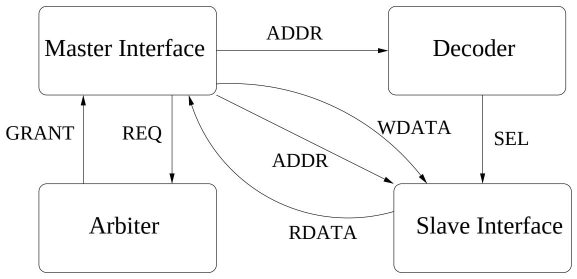

Figure 1: The Virtual Bus

In this paper, we address the first design step. We start with the specification document written with the traditional mixture of English sentences, drawings and timing diagrams. We manually construct a formal model for each generic component and prove each one correct. Then, we model their interconnections and prove the communication correct.
A typical bus [ARM99, IBM01] is composed of four elements: an address decoder, a bus arbiter, slave (or target) and master (or initiator) interfaces (Fig 1). The protocol is mainly of master-slave type, i.e. based on *handshakes*. The handshake (or *point to point connection*) is the basic communication scheme of the AHB bus. The next section presents our modeling and validation approach on this basic scheme. Section 3 introduces the AHB bus. Section 4 shows the modeling and the validation of the address decoder and the bus arbiter. In section 5, the principles exposed in section 2 are applied to the AHB bus. Finally, conclusions are given in section 6.

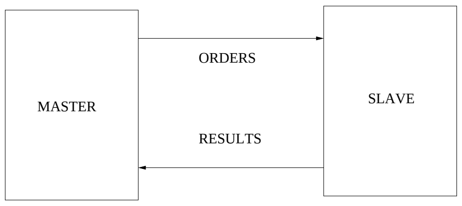

Figure 2: Master/Slave interconnection scheme

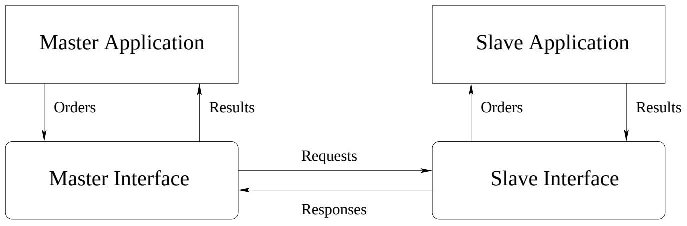

Figure 3: Point to point interconnection scheme

2 Principle of the communication

2.1 Master/Slave communication

Master/Slave communication is the basic protocol for hardware components. In this configuration, drawn on Fig 2, masters send *orders* to slaves, which send *results* as answers. A communication is always started by a master. Typically, masters are processors and slaves memories, so, *orders* are typically *read* or *write* operations to a specific address or block of addresses; *results* are typically data or a block of data and the slave status (*e.g.* no errors, global errors, not ready ...).

Let master M be a processor performing a divide operation between two integers and slave S be a memory containing a set of integers. To perform the divide operation, M first reads the memory to obtain the values of the operands, say a and b. Then, it uses a given algorithm *divide_alg* to compute the integer division of a by b. Finally, it writes the result, say c, in the memory. We see here, as it was introduced in [RSV97], that the communication operations (read and write orders) are *orthogonal* to the computation operation (the divide algorithm). These two kinds of operations are separated in two types of components: the interfaces and the applications. The point to point communication scheme with interfaces is drawn on Fig. 3. To make the difference between interface-application and interface-interface communication clear, the former dialogue is denoted by *results* and *orders*, the latter by *requests* and *responses.* We represent *orders* and *results* by the following lists:

- order = (O L [D]), where O is an operation (*i.e.* read, write), L is a location and D is the data to be written. [D] is optional for read operations.
- result = (status) or (status data)

Assume requests and responses are represented by the following list:

- request = (R/W addr [Data]), R/W = 1 means a read operation at the address **addr** is requested, R/W = 0 means a write operation of **Data** at the address **addr** is requested, the [Data] is optional for read operations.

- response = (status) or (status data)

We distinguish integers a and b from their memory addresses $@a$ and $@b$. The master application performs the following operations:

1. a = (read @a) then wait for a result
2. b = (read @b) then wait for a result
3. (write @a (divide_alg a b))

The communication events are the following. First, the master interface receives the read order and produces the request *(1 @a)*. The slave interface receives this request and transmits the order *(read @a)* to the slave application. This application effectively reads its database and returns the value a stored at the address $@a$ through the result *(OK a)* which is successively transmitted back to the master interface, and to the master application, which is now able to send the next order. The events for the second read operation are similar. For the write operation, let us consider that $(divide\_alg\ a\ b) = c$. The master interface receives the write order and produces the request *(0 @a c)*. The slave interface receives this request and computes the order *(write @a c)*. The result *(OK)* follows the same way as the read result from the slave application to the master application.

Now that we have seen the information flow, we present our modeling approach in the next section.

2.2 Functional modeling

Our formalization is functional, *i.e.* each component and each transfer is modeled by a function. Time is abstracted away, and functional composition is used to express sequential events.
To formalize the previous example of the divide program, we start with the interfaces (Fig. 4). The master interface has two inputs (the order and the response) and two outputs (the request and the result), so it is represented by a function of two parameters returning two objects. Each input and output is in fact a tuple. Practically, there are two ways to model a tuple: either we consider the tuple as one input (or output), or each element of the tuple is one input (or output). In the present paper, we consider that each element of a tuple is an input, because it is closer to the hardware notion of signal. Concerning the master interface, each input is a formal of the function and the output is a list of two lists. The first one contains the signals connected to the slave interface, and the second one the signals connected to the master application. The definition of this function is:

```
(defun master_interface (O L D ST SD)
  (if (equal O 'read)
      (list (list 1 L D) (list SD ST))
      (list (list 0 L D) (list SD ST))))
```

To make connection of functions clear, we define an accessor function for each output of the interface. The accessors for the master interface are:

- (R/W x) = (nth 0 (nth 0 x))
- (ADDR x) = (nth 1 (nth 0 x))
- (Data x) = (nth 2 (nth 0 x))
- (S-master x) = (nth 1 (nth 1 x))

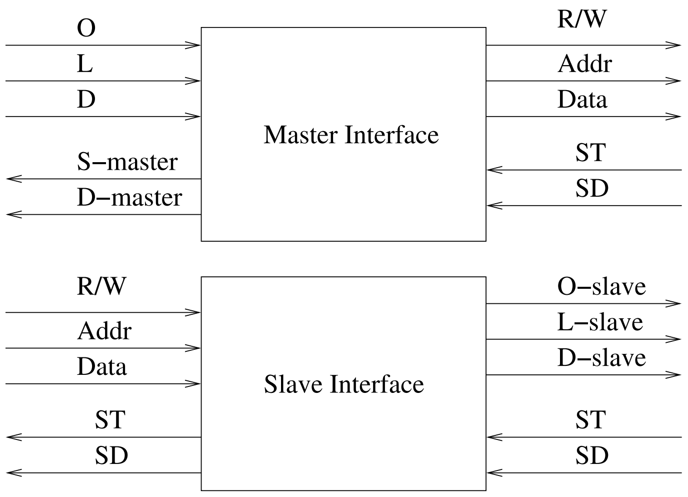

Figure 4: Definition of master and slave interfaces

- (D-master x) = (nth 0 (nth 1 x))

The slave interface is drawn on Fig. 4 and modeled using the same approach:

```
(defun slave_interface (R/W Addr Data  SD ST)
   (if (equal R/W 1)
       (list (list 'read Addr Data) (list SD ST))
       (list (list 'write Addr Data) (list SD ST))))
```

The accessors are:

- (O-slave x) = (nth 0 (nth 0 x))
- (L-slave x) = (nth 1 (nth 0 x))
- (D-slave x) = (nth 2 (nth 0 x))
- (SD x) = (nth 0 (nth 1 x))
- (ST x) = (nth 1 (nth 1 x))

These two interfaces are composed together (*i.e.* the components are connected) yielding two functions: one representing a transfer from the master interface to the slave interface, and the other one a transfer from the slave interface to the master interface. A transfer from the master interface to the slave interface is defined as the result of the *slave_interface* function applied to the result of the *master_interface* function; so, by the following composition:

$$trans\_M\_to\_S\;(O\;L\;D) \;=\; slave\_interface \circ master\_interface\;(O\;L\;D) \qquad (1)$$

which gives the following ACL2 function:

```
(defun trans_M_to_S (O L D SD ST)
  (slave_interface ; R/W signal
                  (R/W (master_interface O L D SD ST))
```

```
   ; ADDR signal
   (ADDR (master_interface O L D SD ST))
   ; Data signal
   (Data (master_interface O L D SD ST))
   ; resulting Data(D) and Status (S) given by the slave application
                    SD ST))
```

Similarly, a transfer from the slave interface to the master interface is defined as the composition:

$$trans\_S\_to\_M \; (D \; S) \; = \; master\_interface \; \circ \; slave\_interface \; (D \; S) \tag{2}$$

which gives the following function:

```
(defun trans_S_to_M (O L D SD ST)
  (master_interface  O L D ; not considered
     ; to select the data of the response
     (SD (slave_interface O L D SD ST))
     ; to select the status of response
     (ST (slave_interface O L D SD ST))
))
```

The slave application is a memory, and we model it by a list named *memo*. The read and write operations are performed through calls of the *nth* and *put-nth* ACL2 functions. The function representing the slave application is:

```
(defun slave_application (O L D memo)
   (if (equal O 'read)
       (list (nth L memo) 'OK)
       (list (put-nth L D memo) 'OK))); write order
```

Without the slave application, it is not possible to compose Equations 1 and 2. But, using the *slave_application* function, we can define a *transfer* function modeling the complete execution of an order, *i.e.* transmitting the order and getting the result back. A transfer is represented by the following composition:

$$trans\_order \; (O \; L \; D) \; = \; trans\_S\_to\_M \; \circ slave\_application \; \circ \; trans\_M\_to\_S \; (O \; L \; D) \tag{3}$$

which can be paraphrased as follows. First, transmit the order to the slave application, then effectively compute the corresponding operation and finally transmit the result to the master application. We get the ACL2 function:

```
(defun trans_order (O L D memo)
  (trans_S_to_M nil nil nil
                (nth 0 ; to select the returned data
                     (slave_application
                                       ; operation O
                      (O-slave (trans_M_to_S O L D nil nil))
                                       ; location L
                      (L-slave (trans_M_to_S O L D nil nil))
                                       ; data D
                      (D-slave (trans_M_to_S O L D nil nil))
                      memo))
                (nth 1 ; to select the returned status
                    (slave_application
                                        ; operation O
                     (O-slave (trans_M_to_S O L D nil nil))
                                        ; location L
                     (L-slave (trans_M_to_S O L D nil nil))
                                        ; data D
                     (D-slave (trans_M_to_S O L D nil nil))
                    memo)))))
```

Suppose *(divide_alg a b)* computes the division of a by b, the master function is:

```
(defun master_function (a_address b_address c_address memo)
  (trans_order 'write c_address
               (divide_alg (D-master (trans_order 'read a_address nil memo))
                           (D-master (trans_order 'read b_address nil memo)))
               memo))
```

which completes the model of the point to point connection.

2.3 Proving essential properties

As for any verification process, we need to define a reference. For the point to point communication scheme, it is a direct communication between master and slave applications (see Fig. 2, page 2). The underlying principle is that the introduction of interfaces do not modify the communication function between the applications. With no interfaces, if a master application sends a (read L d) or a (write L D) order, the slave application receives a (read L D) or a (write L D) order. With interfaces, the *trans_M_to_S functions*, modeling the order transmission, applied to a (read L D) or a (write L D) order should return a (read L D) or a (write L D) order. Theorems 1 and 2 below, proven by the definition of *master_interface* and *slave_interface* state it.

Theorem 1 *(Correctness of the read transmission)*

```
(defthm trans_M_to_S_read
  (implies (equal O 'read)
           (and (equal (O-slave (trans_M_to_S O L D SD ST)) 'read)
                (equal (L-slave (trans_M_to_S O L D SD ST))  L)
                (equal (D-slave (trans_M_to_S O L D SD ST))  D))))
```

Theorem 2 *(Correctness of the write transmission)*

```
(defthm trans_M_to_S_write
  (implies (equal O 'write)
           (and (equal (O-slave (trans_M_to_S O L D SD ST)) 'write)
                (equal (L-slave (trans_M_to_S O L D SD ST))  L)
                (equal (D-slave (trans_M_to_S O L D SD ST))  D))))
```

Let us consider the function *trans_S_to_M* that transmits a result. Because the result has the same structure for read and write operation, there is only one theorem, proven using the definition of *master_interface* and *slave_interface.* This theorem states that if a slave application sends a (RD RS) result to the slave interface, the master application receives the same (RD RS) result.

Theorem 3 *(Correctness of the result transmission)*

```
(defthm trans_S_to_M_thm
  (and (equal (D-master (trans_S_to_M O L D SD ST)) SD)
       (equal (S-master (trans_S_to_m O L D SD ST)) ST)))
```

These three theorems prove that the communication between the applications has the same behavior *with and without* the interfaces.
Let us now consider the *trans_order* function. As there are two different operations, we have two theorems. The first one proves that the read operation behaves correctly, the second one proves the write operation. The first theorem states that the data returned by a call to *trans_order(read L D memo)* is equal to a direct read (a call of nth) in memo:

Theorem 4 *(Correctness of a read order)*

```
(defthm trans_order_read
  (implies (equal O 'read)
           (equal (D-master (trans_order O L D memo))
                  (nth L memo)))
  :hints (("GOAL" :in-theory (disable nth D-master))))
```

Proof. Proof is obtained by theorems 1, 2, 3 and the definition rule slave_application. The hint is here of importance, because we consider *(nth L memo)* as a "token" and we do not want it be expanded. Q.E.D.
The second theorem (proven like the one above) states that a write order is equal to a "put-nth" on memo:

Theorem 5 *(Correctness of a write order)*

```
(defthm trans_order_write
  (implies (equal O 'write)
           (equal (D-master (trans_order O L D memo))
                  (put-nth L D memo)))
  :hints (("GOAL" :in-theory (disable nth D-master))))
```

If the *master_application* function is correct, we should prove, once memo has been modified by a call of *master_application*, that the data located at *c_address* in *memo* is equal to the division of the integers located at *a_address* and *b_address* in *memo*. So, we prove the theorem below.

Theorem 6 *(Correctness of the master application)*

```
(defthm divide_thm
  (implies (and (< c_address (len memo))
                (< 0 c_address))
           (equal
            (nth c_address
                 (D-master
                  (master_function a_address b_address c_address memo)))
            (floor (nth a_address memo) (nth b_address memo))))
  :hints (("GOAL" :in-theory (disable D-master floor nth Data))))
```

Proof. Obtained by theorem 1, 2 and the "nth-put-nth" lemma of the list-defthms book. Q.E.D.

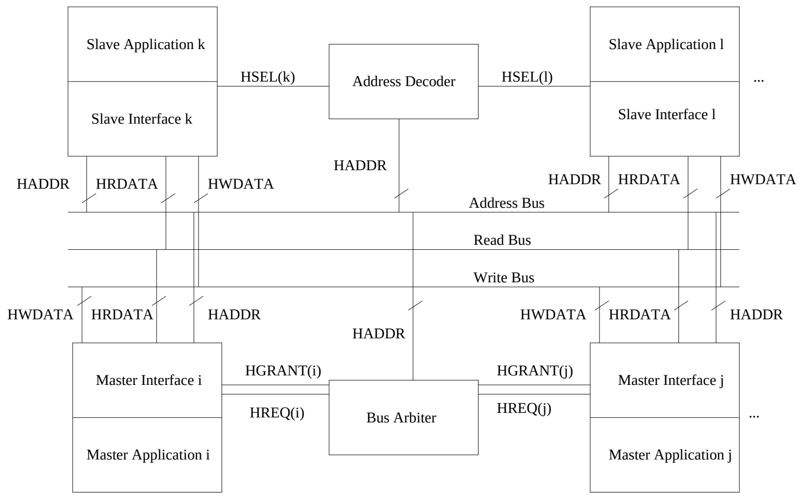

Figure 5: Organization of the AMBA AHB architecture

3 The AMBA AHB bus system

Arm processors are widely used in SoCs and embedded systems, so is the *Advanced Microcontroller Bus Architecture (AMBA)* and particurlarly the *Advanced High-performance Bus (AHB)*. Its global structure is drawn on Fig. 5.

The AHB on chip bus allows the interconnection of **n** masters (typically: processing units) and **m** slaves (typically: memory units), where **n** and **m** are parameters. There are three different buses: **HRDATA** conveying data to be read, **HWDATA** for data to be written, and **HADDR** for the addresses. The communication involves four components:

1. a decoder: it receives an address and activates the corresponding slave
2. an arbiter: it reveives requests and grants the bus to a unique master
3. master interfaces: initiate transfers
4. slave interfaces: answer to master requests

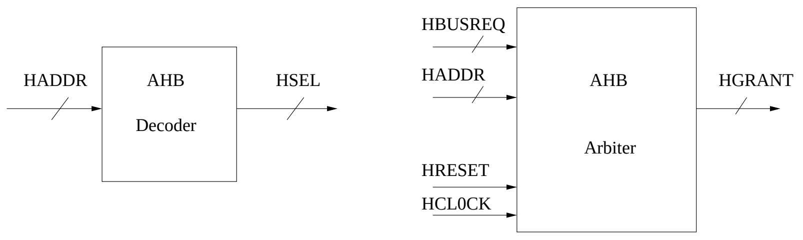

Figure 6: (Simplified) Schemes of the decoder and the arbiter

The decoder (Fig 6) is used to select the slave that owns the required data. Address decoding depends on memory organization, and details about it are given in section 4.1. The decoder

activates a slave x by setting the x'th bit of the **HSEL** signal to 1 and the others to 0.
The role of the arbiter (Fig 6) is to grant bus access to one master requesting it. The arbiter is unique in a given architecture and uses a priority scheme to select a master. This algorithm is not specified in [ARM99] but must obey some rules. The most important is to preserve the *mutual exclusion* of bus accesses. The AHB architecture is not a tri-states bus, so a default master is granted when no master requests the bus. We select master number 0. The slave interface (Fig 7)

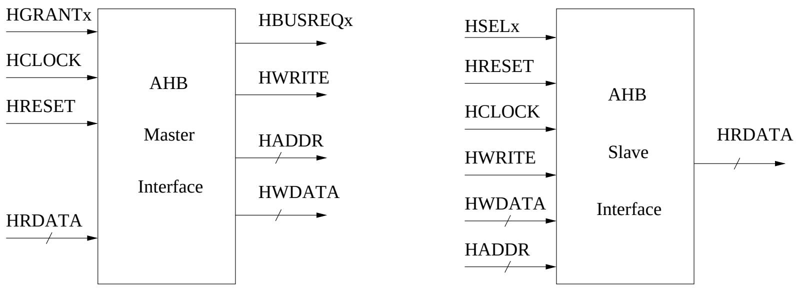

Figure 7: (Simplified) Schemes of the master and slave interfaces

uses the **HSELx** signal to determine if it is active or not. If **HSELx** is high, the slave answers the master read or write request in one cycle, or it splits the transfer. The master interface (Fig 7) produces all the signals needed for a transfer.

4 Memory and bus arbitration

In this section, we formalize the two main components of the bus protocol (the address decoder and the bus arbiter) and we prove them correct with respect to [ARM99]. As we deal with functional models, reset and clock signals are ignored in our formal representation.

4.1 Address decoding

Let **Card_S** be the number of slaves, a generic parameter of the decoder of type natural. The address decoding corresponds to the selection of an element of a set. The decoder function associates a set of addresses to a slave. As we abstract addresses to naturals, the input domain of the decoder function is the set of naturals. The output domain is the interval $[\,0\,,\,Card\_S\,-\,1\,]$. S_i denotes the slave number i, $i\,\in\,[\,0\,,\,Card\_S\,-\,1\,]$.
Each slave is connected to a memory of **MEM_SIZE** addresses, and the global system memory is the product $Card\_S\,\times\,MEM\_SIZE$. During transfers, the master puts the global address, which ranges from 0 to $Card\_S\,\times\,MEM\_SIZE\,-\,1$, on the **HADDR** bus, and the slave selected by the decoder reads or writes the data to its local address **UNADDR**. A data has consequently two addresses - a global and a local address- related by the following equation:

$$UNADDR \,=\, HADDR\, mod\, MEM\_SIZE \tag{4}$$

The slave possessing the data at the HADDR address is S_i, where i is computed by the equation:

$$i \,=\, \frac{HADDR}{MEM\_SIZE} \tag{5}$$

The decoder is modeled by the functions *select(Card_S SEL)* and *decoder(MEM_SIZE, Card_S, ADDR)* below:

```
(defun  select (Card_S SEL)
  (cond ((not (integerp Card_S)) nil)
        ((<= Card_S 0) nil)
        ((equal SEL 0)
         (cons 1 (select (1- Card_S) (1- SEL))))
        (t
         (cons 0 (select (1- Card_S) (1- SEL))))))

(defun decoder (MEM_SIZE Card_S HADDR)
  (select Card_S (floor HADDR MEM_SIZE)))
```

Function *select* is recursive over **Card_S**, the number of list elements. It returns the empty list if **Card_S** is not a natural integer. Otherwise, it concatenates 1 (if **sel** $= 0$) or 0 (if **sel** $\neq 0$) to the list resulting from the recursive call of *select* over $Card\_S - 1$ and $sel - 1$. Thus *select* constructs a list of **Card_S** elements, all equal to 0 except the *sel'th* one, provided $0 \leq sel \leq Card\_S$. The main property to prove on these functions is that they select only one slave possessing the desired data. Clearly, once the proof is made on *select*, the proof on *decoder* is straightforward due to equation 5. We prove the following theorems.

Theorem 7 *(Selection of the right slave)*

```
(defthm ith_select_=_1
  (implies (and (integerp i) (integerp Card_S)
                (>= i 0) (> Card_S i))
           (equal (nth i (select Card_S i )) 1)))
```

Proof. We first prove a lemma stating the property is true if $i = 0$. This lemma could be avoided but eases the proof. ACL2 proves the theorem automatically by induction over $Card\_S$ and i. Q.E.D.

Theorem 8 *(Uniqueness of the selection)*

```
(defthm pth_select_=_0
  (implies (and (integerp p) (integerp Card_S)
                (<= 0 p) (< p Card_S)
                (not (equal p i)))
           (equal (nth p (select Card_S i)) 0))
  :hints (("GOAL"
           :induct (function_hint_th2_select p Card_S i))))
```

Proof. As for theorem 1, we prove a lemma stating the first element of *select* is 0 when $i \neq 0$. We prove another lemma stating *select* returns a cons-pair. The induction scheme suggested by *(select Card_S i)* is over $Card\_S$ and i. Here, we need an induction hypothesis on Card_S, i and p. We define a function called *function_hint_th2_select* that suggests the needed induction scheme and ACL2 proves the theorem automatically. Q.E.D.

4.2 Bus arbitration

The bus arbiter determines which master should be granted bus access, using a priority scheme modeled by a priority matrix (Fig 8). The number P of lines sets the number of priority levels and the number N of columns is the number of masters having the same priority. Lines and columns are numbered from 0. The matrix elements correspond to a single master, and the master number

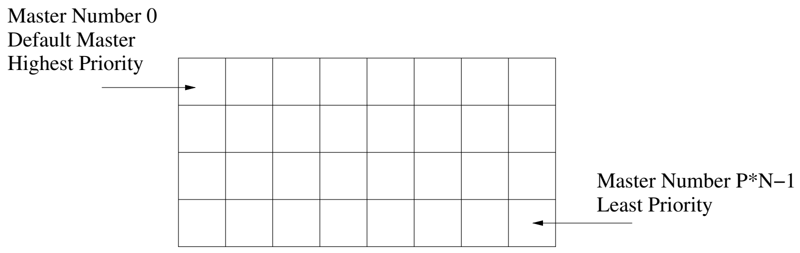

Figure 8: The matrix modeling the priority scheme

is computed from line l and column c by: master_num = l*N + c. Master number 0, the default master, has the highest priority. The input domain of the arbiter function is the set of NxP bit matrices **MREQ** representing the master requests. The output domain is the set of bit vectors of length NxP where only one bit is 1. The function uses the following algorithm to select the master that should be granted bus access:

- determine the first line **RLINE** containing at least one request *(Stage_P)*
- determine the next requesting master to be granted the bus, according to a round robin scheme on the line *(round_robin)*
- compute the number of this master *master_num* from the line number and build the output list **HGRANT** where bit number master_num is 1 *(master_num and arbiter)*

In order to model these functions we need to define some predicates. The predicate *no_requestp_matrix(MREQ)* recognizes a matrix containing no request, *i.e.* all its elements are 0. The predicate *no_requestp(L)* recognizes a line with no request. So, the first line containing at least one request is the first element of the priority matrix MREQ such that *no_requestp(car MREQ)* does not hold. The number of this stage is computed by the function *stage_P(MREQ)* below:

```
(defun stage_P (MREQ)                       ; returns the line number
  (cond ((endp MREQ) 0)                     ; for the highest priority request
        ((no_requestp_matrix MREQ) 0)       ; if empty list or no request returns 0
        ((not (no_requestp (car MREQ))) 0); we count the number of stages
        (t                                  ; containing no request until we meet
         (+ 1 (stage_P (cdr MREQ))))))      ; a stage with at least one request
```

We consider the function correct if prior stages to the returned one contain no request, and if the returned stage in non-empty. We prove the following theorems:

Theorem 9 *(Respect of the priority scheme)*

```
(defthm prior_scheme                        ; we prove that each stage j prior
  (implies (and (equal (stage_P MREQ) i)    ; to the returned one i contains
                (< j i) (<= 0 j)            ; no request
           (no_requestp (nth j MREQ))))
```

Proof. The proof is made by induction over j and MREQ. The proof is fully automatic and ACL2 uses type-prescription rules of the functions *no_requestp*, *no_requestp_matrix* and *stage_P*. Q.E.D.

Theorem 10 *(Chosen stage is not empty, provided there is at least one request)*

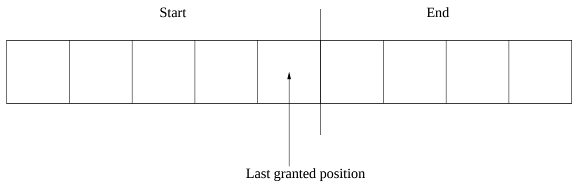

Figure 9: Circular traversal of a line of the priority matrix

```
(defthm chosen_stage_not_empty
  (implies (and (equal (stage_P MREQ) i) (not (no_requestp_matrix MREQ)))
   (not (no_requestp (nth i MREQ)))))
```

Proof. The proof is made by induction over MREQ and uses the same type prescriptions rules as for theorem 9. Q.E.D.

The next important step is the computation of the next master of the returned stage. On a given line, masters have the same priority, and the access policy is “round robin”. A circular traversal of the stage is performed from the last granted position to the next requested position (Fig. 9).
We split a stage after the last granted position; if the end part contains at least one request, the circular traversal starts with the first element of this part. Otherwise, the traversal begins with the first element of the stage. The “start” part is obtained with the function *firstn(n L)* that returns the first n elements (without the *n'th*) of L; the “end” part is obtained with the function *lastn(n L)* that returns the last n elements (with the *n'th*) of L. We define the function *find_next_1(L)* that returns the position (starting from 0 and counted from the first element of L) of the first 1 met in L. The computation of the next master in a given stage RLINE is represented by the following function:

```
(defun round_robin (RLINE Last_Granted); RLINE is (nth (stage_P MREQ))
  (cond ((no_requestp RLINE) 0)        ; if the end part of the line contains
        ((no_requestp (lastn (1+ Last_Granted) RLINE)); no request, find_next_one
         (find_next_1 (firstn (1+ Last_Granted) RLINE))); computes the position
        (t                            ; of the next 1 in the first part of the line
         (+ (1+ Last_Granted)         ; else we proceed in the last part
            (find_next_1 (lastn (1+ Last_Granted) RLINE))))))
```

The main property that should hold is that there is effectively a round robin. In particular, assume the master (number k) which was last granted access to the bus still requests access the next time the line is selected, and one or more other requests are also present on that line, then the next bus access is not granted to master number k. This is expressed by the following theorem:

Theorem 11 *(No Deadlock)*

```
(defthm no_deadlock
  (implies (and (integerp i) (<= 0 i)
                (equal (nth Last_Granted RLINE) 1) (list_of_1_and_0 RLINE)
                (not (equal Last_granted i)))
           (implies (equal (nth i RLINE) 1)
                    (not (equal (round_robin RLINE Last_Granted) Last_Granted))))
  :hints (("GOAL" :use (:instance lemma1_no_deadlock)
                  :in-theory (disable lemma1_no_deadlock firstn)))
  :rule-classes ((:rewrite :match-free :all))
)
```

Proof. The definition of *round_robin* suggests three cases. The hypothesis *(equal (nth i RLINE) 1)* is in contradiction with the first one. The third case is obvious. The remaining case is proven by the lemma used in the hint. This lemma proves the theorem if i is lower than $1 + \mathit{Last\_Granted}$. Q.E.D.

Now, let **E** be the stage number, **N** its length and **New** the position in the stage of the new bus owner. The master M_i should be granted bus access, where i is computed according to the following equation:

$$i = New + N \times E \tag{6}$$

This is done by the function below:

```
(defun master_num (MREQ N Last_Granted)
  (+ (* (stage_P MREQ) N)
     (round_robin (nth (stage_P MREQ) MREQ) Last_Granted)))
```

The last step is to build the output vector HGRANT that selects the right master. This is done using the function *select* defined for the decoder:

```
(defun arbiter (N P MREQ Last_Granted)
      (select (* N P) (master_num MREQ N Last_Granted)))
```

On this function, we prove the uniqueness and the correctness of the selection by proving two theorems similar to those proven on the *select* function. For instance, the theorem similar to theorem 7 is :

Theorem 12 *(Selection of the right master)*

```
(defthm nth_arbiter_=_1
  (implies (and (integerp N) (< 0 N) ; there is at least one master unit
                (integerp Last_Granted) (<= 0 Last_granted)
                (integerp P)
                ; P is the number of line in the priority matrix
                (equal P (len MREQ))
                ; N the number of elements of each line
                (equal (len (car MREQ)) N)
                (not (no_requestp_matrix MREQ)) ; there is at least one request
                (uniform_listp MREQ) ; each line of MREQ has the same length
                ; the last_granted master has a valid number
                (< (1+ Last_granted) N)
                (consp MREQ)
                (consp (cdr MREQ))
                ; the returned line contains bits
                (list_of_1_and_0 (nth (stage_P MREQ) MREQ))
)
   (equal (nth (master_num MREQ N Last_granted)
                      (arbiter N P MREQ Last_granted)) 1))
  :hints (("GOAL" :use (:instance master_num_<_P*N)
                  :do-not-induct t
                  :in-theory (disable  master_num_<_P*N
                                       DISTRIBUTIVITY
                                       ))))
```

Proof. We prove that *master_num* returns a positive integer less than the number of existing master units $P \times N$, then the hypotheses of theorem 7 can be relieved. Q.E.D.

5 Validation of the communication

Any communication on the bus takes place between one master and one slave. The proofs of correctness of the decoder and the arbiter mean that these two components are correctly chosen among an arbitrary number of units. Here, correctly means that the master involved in the communication owns the bus, and the slave the required locations of data. So, we reduce the bus communications to a "generic" point to point scheme and are back to the case exposed in section 2. So, to have a similar model, we add the signals between interfaces and applications (orders and results) to the AHB interfaces. The signals *O, L, D, D-master* are added to the AHB master interface, and the signals *O-slave, D-slave, L-slave, SD* are added to the AHB slave interface. The signals between AHB interfaces are those of section 2, but with new names, which are:

- R/W = HWRITE
- ADDR = HADDR
- Data = HWDATA
- ST is not used anymore (always "OK" in the modeled slave application)
- SD = HRDATA

5.1 Interfaces modeling

The slave interface is active when the HSEL signal is high. The slave interface function returns a list of two lists of signals if HSEL is equal to 1, and simply *nil* otherwise. The first list contains the data sent to the master application as a response. As the status of our slave application (presented in the second section) is always "OK" we do not model it. The second list contains the signals given by the master application and sent to the slave application. So, the interface function is very simple; in fact its task is to transfer signals and to compute the local address. The ACL2 code of the slave interface function is:

```
(defun slave_interface (HSEL HWRITE HADDR HWDATA SD MEM_SIZE)
  (if (equal HSEL 1)
      (list (list (if (equal HWRITE 0)
                      'read
                      'write)
                  (mod HADDR MEM_SIZE) HWDATA)
            (list SD))

      nil))
```

As in section 2, we define accessors for the slave interface outputs:

- (O-slave x) = (nth 0 (nth 0 x))
- (L-slave x) = (nth 1 (nth 0 x))
- (D-slave x) = (nth 2 (nth 0 x))

- (HRDATA x) = (nth 0 (nth 1 x))

The master interface function is active when the HGRANT signal is high. The master interface function returns a list of two lists of signals if HGRANT is equal to 1, and *nil* otherwise. The function is very simple; its task is to transmit signals and to assign the HWRITE signal to 1 in case of a read transfer and to 0 in case of a write transfer. The ACL2 code of the master interface function is given below:

```
(defun master_interface (O L D HRDATA HGRANT)
  (if (equal HGRANT 1)
      (list (list (if (equal O 'Read)
                      1
                      0)
                  L
                  D)
            (list HRDATA))
      nil))
```

We define the following accessors for the master interface outputs:

- (HWRITE x) = (nth 0 (nth 0 x))
- (HADDR x) = (nth 1 (nth 0 x))
- (HWDATA x) = (nth 2 (nth 0 x))
- (D-master x) = (nth 0 (nth 1 x))

5.2 Transfers modeling

For the AHB bus, transfers are modeled using compositions similar to those proposed in section 2. The principal difference consists in the addition of the decoder and the arbiter, and thus of the HSEL and HGRANT signals. In fact, this just adds one input to the master interface function and one input to the slave interface function. The ACL2 code of the *trans_M_to_S* function is:

```
(defun trans_M_to_S (O L D N Card_S P Last_Granted MREQ Slave_Number
                     SD MEM_SIZE)
  (slave_interface
   (nth Slave_Number
        (decoder MEM_SIZE Card_S
                 (HADDR
                  (Master_interface O L D SD
                                    (nth (master_num MREQ N Last_Granted)
                                         (arbiter N P MREQ Last_Granted))))))
  (HWRITE
   (Master_interface O L D SD
                     (nth (master_num MREQ N Last_Granted)
                          (arbiter N P MREQ Last_Granted))))
  (HADDR
   (Master_interface O L D SD
                     (nth (master_num MREQ N Last_Granted)
                          (arbiter N P MREQ Last_Granted))))
  (HWDATA
(19)  (Master_interface O L D SD
(20)                    (nth (master_num MREQ N Last_Granted)
(21)                         (arbiter N P MREQ Last_Granted))))
(22) SD MEM_SIZE))
```

On lines (3) to (9), the slave interface is connected (through the HSEL signal) to the decoder, which is itself connected - on lines (6) to (9) - to the master interface (through the HADDR signal). The master interface is connected - on lines (7) to (9), (12) and (13), (16) and (17), (19) and (20) - to the arbiter (through the HGRANT signal). On lines (10) to (21), the master interface is connected to the slave interface. Finally, on line (22) we have the data SD given by the slave application and a parameter (the size of the slave memory).
The ACL2 code of the *trans_S_to_M* function is:

```
(1) (defun trans_S_to_M (O L D SD MEM_SIZE Card_S MREQ N P
(2)                      HWRITE HADDR HWDATA Slave_Number Last_granted)
(3)   (master_interface O L D ;not considered in slave to master transfers
(4)                     (HRDATA
(5)                      (slave_interface
(6)                       (nth Slave_Number
(7)                            (decoder MEM_SIZE Card_S L))
(8)                       HWRITE HADDR HWDATA ; not considered
(9)                       SD
(10)                      MEM_SIZE))
(11)                    (nth (master_num MREQ N Last_Granted)
(12)                         (arbiter N P MREQ Last_Granted))))
```

On lines (4) to (6) the HRDATA output of the slave interface is connected to the master interface. The signals O,L,D, HWRITE, HADDR and HWDATA are used in transfers from the master to the slave interface, but they are not involved in transfers from the slave to the master interface, and can therefore be set to a constant value, *'undef*, when one calls *trans_S_to_M* (see for instance theorem 13).

5.3 Transfers validation

To validate the transfers we would like to establish theorems close to theorems 1, 2 and 3 of section 2. The first theorem states that a read order emitted from the master application is correclty received by the slave application. The SD input of *trans_M_to_S* has no importance in this way, and is set to *'undef.*

Theorem 13 *(Correctness of the read transmission)*

```
(defthm trans_M_to_S_thm
  (implies (and
            ; P is the number of priority level(s)
            (integerp P) (equal P (len MREQ))
            ; N is the length of each level
            (equal (len (car MREQ)) N)
            ; at least one master
            (integerp N) (< 0 N)
            ; each level has the same length
            (uniform_listp MREQ)
            ; the last owner has a valid number
```

```
          (integerp Last_Granted) (<= 0 Last_Granted)
          (< (+ 1 Last_granted) N)
          ; at least one request
          (not (no_requestp_matrix MREQ))
          (consp MREQ) (consp (cdr MREQ))
          ; each level is a line of bits
          (list_of_1_and_0 (nth (stage_P MREQ) MREQ))
          ; at least one slave unit
          (integerp Card_S) (< 0 Card_S)
          ; L is a valid address
          (integerp L) (<= 0 L) (< L (* Card_S MEM_SIZE))
          ; the size of the slave memory is at least one
          (integerp MEM_SIZE) (< 0 MEM_SIZE)
          ; the slave is active
          (equal Slave_Number (floor L MEM_SIZE)) ; *hypothesis*
          )
         (and (equal (O-slave
                       (trans_M_to_S O L D N Card_S P Last_Granted MREQ
                           Slave_Number 'undef MEM_SIZE))
                     'read)
              (equal (L-slave
                      (trans_M_to_S O L D N Card_S P Last_Granted MREQ
                           Slave_Number 'undef MEM_SIZE))
                     (mod L MEM_SIZE))
              (equal (D-slave
                      (trans_M_to_S O L D N Card_S P Last_Granted MREQ
                                     Slave_Number 'undef MEM_SIZE))
                     D))))
```

Proof. In the considered bus architecture, every transfer takes place between *one* master and *one* slave, so, we reduce the communication between an arbitrary number of masters and slaves to a point to point connection between a *well* chosen master and a *well* chosen slave. These assumptions are expressed through the *hypothesis* marked above and in lines 8-9, 12-13, 16-17 and 20-21 of the *trans_M_to_S* function. The *hypothesis* relieves the hypotheses of theorem 7 of section 4.1 and we obtain , once *trans_M_to_S* has been expanded, that *(nth slave_number (decoder MEM_SIZE Card_S L)) = 1* , *i.e.* the slave unit is active. When we connect the arbiter, the call *(nth (master_num MREQ N Last_Granted) (arbiter N P ...)* in *trans_M_to_S* (see definition on page 16) and the hypotheses of the above theorems induce that theorem 9 holds, *i.e.* the master owns the bus. The communication is now reduced to a simple communication between the two interfaces and by the functions definitions we prove the theorem. Q.E.D.
The next theorem is proven in a similar way, and deals with a write order:

Theorem 14 *(Correctness of the write transmission)*

```
(defthm trans_M_to_S_write
  (implies (and ...
                same hypotheses as for trans_M_to_S_read
                except that the order is write:
          (equal O 'write)
         )
         (and (equal (O-slave
                     (trans_M_to_S O L D N Card_S P Last_Granted MREQ
                         Slave_Number 'undef MEM_SIZE))
                   'write)
            (equal (L-slave
                    (trans_M_to_S O L D N Card_S P Last_Granted MREQ
                         Slave_Number 'undef MEM_SIZE))
                   (mod L MEM_SIZE))
            (equal (D-slave
                    (trans_M_to_S O L D N Card_S P Last_Granted MREQ
                                   Slave_Number 'undef MEM_SIZE))
                   D))))
```

Finally, we prove:

Theorem 15 *(Correctness of the result transmission)*

```
(defthm trans_S_to_M_thm
  (implies (and ...
                hypotheses of the previous theorems
                without the one on O
                ...
                )

           (equal (D-master
                    (trans_S_to_M O L D SD MEM_SIZE Card_S MREQ N P HWRITE HADDR
                                  HWDATA Slave_Number Last_Granted))
                  SD)))
```

We have proven theorems on a structure containing an arbitrary number of interfaces, one decoder and one arbiter. We can go a little further because every slave application is a memory. A memory is modeled by the following function (which is very close to the one presented in section 2.2):

```
(defun slave_memory (MEMO O UNADDR D)
  (cond ((equal O 'write)
         (list (put-nth UNADDR D MEMO) D))
        ((equal O 'read)
         (list MEMO (nth UNADDR MEMO)))))
```

Using Equation (3), we model a single transfer as follows:

```
(defun single_transfer (O L D N P Card_S Last_Granted MREQ Slave_Number
                        MEM_SIZE MEMO)
  (list
   (trans_S_to_M O L D
              (nth 1
                   (slave_memory MEMO
                     (O-slave
                      (trans_M_to_S O L D N Card_S P Last_Granted
                                    MREQ Slave_Number 'undef MEM_SIZE))
                     (L-slave
                      (trans_M_to_S O L D N Card_S P Last_Granted
                                 MREQ Slave_Number 'undef MEM_SIZE))
                          (D-slave
                           (trans_M_to_S O L D N Card_S P Last_Granted
                                 MREQ Slave_Number 'undef  MEM_SIZE))))
                  MEM_SIZE Card_S MREQ N P O L D
                  Slave_Number Last_Granted)
   (nth 0
        (slave_memory MEMO
                        (O-slave
                           (trans_M_to_S O L D N Card_S P Last_Granted
                                 MREQ Slave_Number 'undef MEM_SIZE))
                        (L-slave
                         (trans_M_to_S O L D N Card_S P Last_Granted
                                 MREQ Slave_Number 'undef MEM_SIZE))
                        (D-slave
                         (trans_M_to_S O L D N Card_S P Last_Granted
                                 MREQ Slave_Number 'undef MEM_SIZE))))))
```

The first part of the list represents the signals received by the master application. The second returns the modified memory and is useful to reason on the memory.
On this function, we first prove that a read operation returns the same data as a direct read in the memory (through a call of *nth*):

Theorem 16 *(Correctness of a read operation)*

```
(defthm single_read_transfer
  (implies (and ...
                hypotheses of trans_S_to_M_thm
                ...
                (equal O 'READ)
                )
            (equal (D-Master
                   (nth 0
                        (single_transfer O L D N Card_S P Last_Granted MREQ
                                Slave_Number MEM_SIZE MEMO)))
                  (nth (mod L MEM_SIZE) MEMO))))
```

Then we prove that if we read the memory after a write operation of a given data, we obtain this data:

Theorem 17 *(Correctness of a write operation)*

```
(defthm single_write_transfer
  (implies (and (equal O 'WRITE)
                ...
                hypotheses of trans_S_to_M_thm
                ...
; the size of MEMO is MEM_SIZe
(equal (len MEMO) MEM_SIZE)
                )
           (equal (nth (mod L MEM_SIZE)
                       (nth 1
                    (single_transfer O L D N Card_S P Last_Granted MREQ
                       Slave_Number MEM_SIZE MEMO)))
        D)))
```

This concludes the proof of our model *single_transfer*. The proof of this function could now be used to prove correct master applications, like the divide program of section 2.

6 Conclusion

The formalization involves around twenty functions, more than sixty theorems and the total proof time, on a 400 Mhz bi-processors SUN server, is about thirty seconds. As we model the bus using lists, we use the books *list-defuns* and *list-defthms*. We also use the books *arithmetic/top* and *floor-mod*. More details on the AMBA bus and its functional model can be found in [SB03], with some differences in the details of the ACL2 code, which has since been clarified and extended. This work should be extended to capture pipelined and out-of-order transfers. Finally, we aim at verifying hardware designs and these should be tested against our formal specification.

References

[ARM99] ARM. *AMBA Specification*, 1999.

[IBM01] IBM. *128-bit Processor Local Bus, Architecture Specifications Version 4.4*, 2001.

[RSV97] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interface-Based Design. In *Design Automation Conference*, pages 178–183, 1997.

[SB03] J. Schmaltz and D. Borrione. Formalization and Verification of the AMBA-AHB Communication Architecture Using the ACL2 Theorem Prover. March 2003. Research Report TIMA-RR-03/03-01-FR, http://tima.imag.fr/publications/files/rr/fva_170.pdf(A short version of this report was published in the proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'03), pp 93-100, Poznan, April 2003).

[VSI01] VSI Alliance. *Virtual Component Interface Standard Version 2 (OCB 2 2.0)*, April 2001.