Specification of an Asynchronous On-Chip Bus

Juha Plosila and Tiberiu Seceleanu

University of Turku, Dept. of Information Technology,
Lab. of Electronics and Communication Systems,
FIN-20014 Turku, Finland, tel: +358-2-3336954, fax: +358-2-3336950,
email: { Juha.Plosila, Tiberiu.Seceleanu }@utu.fi

Abstract. The latest improvements in the technology of digital devices allow designers to build whole systems on a single silicon chip. New problems arise in this context, one of them being the complexity of interconnections. Optimizing interfaces has become a tedious design step. Other problematic issues are global clock signal distribution and design composability, for which asynchronous design methodology proves to be a good solution. Formal methods can be used to verify the logical correctness of digital hardware. These methods are well featured for asynchronous designs and this study introduces bus-modeling aspects in the formal framework of Action Systems.

1 Introduction

Modern deep sub-micron silicon technologies have given a real boost to system-on-chip (SOC) design research and development. A key issue in integrating a whole system into a single chip is to realize efficient and reliable interconnections between system modules on the chip, because the overall performance of the system is constrained by the properties of the interconnect. Furthermore, the time required to complete a design task depends strongly on the complexity of the different interconnections in the system. Optimizing interfaces has become a cumbersome process [9].

A common way to implement a digital system is to build it around a *bus* shared by the system modules. As the modules connected to the bus have uniform interfaces, the bus-based design approach offers a relatively rapid method to construct large systems-on-chip. In future high-performance systems, maintaining global synchrony will be increasingly difficult, if not impossible. A solution to this problem comes from the employment of asynchronous, or self-timed, design methodologies [3]. Enhanced composability is another advantage of self-timed approaches over synchronous ones. Therefore, the present study describes a self-timed bus structure. This means that all the components of a system employing the bus interact via handshakes on asynchronous communication channels, while locally they can operate synchronously, controlled by some local clock signal(s).

The integration of a complex digital system requires comprehensive know-how of verification and development of SOC solutions, both at the functional (behavioral) as well as at the physical levels. Because a system can be viewed as a composition of concurrent processes, formal methods of concurrent programming can be used to verify mathematically the logical correctness of digital hardware. The *Action Systems* framework is one of such formalisms. It has recently been applied to the area of asynchronous (self-timed) and synchronous VLSI design [8, 10].

In this paper, we present an Action Systems-based specification of an on-chip bus targeted for SOC designs. While bus-related protocols have been studied before from a formal point of view [6, 7], our study goes one step further and describes the components of a bus-based system. The specification is composed of the formal descriptions of the bus components, including the central bus controller (arbiter) and the abstract models of the master and slave modules attached to the bus. The generality of the specification allow for further derivation towards concrete descriptions, at lower levels of abstraction, following precise rules. We only provide template-components, which can fit a large range of actual devices. From a more specific point of view, the integration of a bus model into our Action Systems framework is a part of our current work on developing a formal design flow for complex systems-on-chip. In this flow, a design process from a specification to an implementable model is viewed as a sequence of correctness preserving refinement steps.

Instead of trying to develop a completely new bus specification, we prefer to adapt an existing standard bus to our needs. The *AMBA bus* specification [1] is an established, open bus standard that serves as a framework for SOC designs. We use many characteristics of this specification in our descriptions. However, mostly because AMBA is a *synchronous* bus, our *asynchronous* bus does not require some of the control issues described in [1]. On the other hand, we introduce several new control signals which are needed in asynchronous communication between system modules. Thus, our representation of the bus controller and the associated masters and slaves would show some differences concerning the interfaces with respect to the AMBA specification. For the matched signals, however, we will use the same notations as in [1].

2 Action Systems

The Action Systems formalism is based on an extended version of Dijkstra's language of *guarded commands* [5]. An *action* A is defined (for example) by

$$\begin{array}{lll} A ::= & x := x'.R & \text{(nondeterministic assignment)} \\ & A_1 \,[\!]\, A_2 & \text{(nondeterministic choice)} \\ & A_1;\ A_2 & \text{(sequential composition)} \\ & P \rightarrow A_1 & \text{(guarded action)} \\ & A_1 \ // \ A_2 & \text{(prioritized composition)} \end{array}$$

where P and R are predicates, x is a variable or a list of variables, and A_1 and A_2 are actions. Semantically, an action A is defined by the *weakest precondition* for A to establish some post-condition Q, denoted $\mathrm{wp}(A, Q)$. The *guard* of an action, gA is defined by $gA \ \widehat{=} \ \neg\mathrm{wp}(A, \mathit{false})$. An action is said to be *enabled*, if its guard is *true*, *disabled* otherwise. Actions are considered *atomic*, meaning that whenever one is selected for execution, it will be completed without interference.

```
sys 𝒜(interface list) ::
|[
 var          local variable declaration
 const        local constant declaration
 expressions  expressions used to ease the readability
 actions      action list
 init         initialization of variables
 do           action composition od
]|
```

Fig. 1. A partial action system representation.

An *action system* $\mathcal{A}$ is an iterative composition of actions. It has the form shown in Figure 1 [11]. Here, the *interface list* defines the global variables through which $\mathcal{A}$ communicates with other systems. The **var** and **const** clauses define the local variables and constants, visible only within $\mathcal{A}$. The **expressions** clause defines shorthand notations for some expressions used within $\mathcal{A}$. The items declared here are evaluated every time they are met during the execution of the system. The **actions** clause describes the atomic actions present in the system. A unique name is given for each action. An action can also be, partly or completely, composed of other actions defined in the **actions** clause. For instance: $A_4 \ : \ (A_1;\ A_2) \ [\!] \ A_3$. Here, the actions A_1, A_2, A_3 are called *included actions* with respect to the action A_4.

In the **init** clause, all the local and interface variables are initialized. For the latter, special care has to be devoted, as these variables have to be initialized with the same values in all the systems that share them.

The system's **do-od** loop contains a composition of the actions defined in the **actions** clause. The composition can be realized using the atomic operators ‘ $[\!]$ ’ and ‘ // ’ and the *macro operators* such as the non-atomic sequential composition operator ‘ **;** ’ which is actually defined in terms of the non-deterministic choice (‘ $[\!]$ ’) and an auxiliary local variable. In the loop, one enabled action is executed at the time. Parallel behavior is modeled by two or more actions that are enabled simultaneously and can be executed in any order without affecting the result of the computation.

Notation. A substitution operation within an action A is denoted by $A[e_{new}/e_{old}]$, where e_{old} refers to an element (variable, predicate, component action etc.) of the original action A, and e_{new} denotes the new element which replaces e_{old} in A. The same notation may also be used for action systems.

Composing action systems. The *parallel composition* of two action systems $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted $\mathcal{A}_1 \parallel \mathcal{A}_2$, is defined as an action system whose loop has the form **do** $A_1 \ [\!] \ A_2$ **od**, where A_1 and A_2 represent the action compositions within the loops of the constituent systems $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. The composed

system merges the global variables of the components keeping the local variables and action identifiers distinct.
Quantified composition. Any composition operator can be *quantified*. This applies to the different types of action compositions and the parallel composition of action systems. The notation is defined as follows:

$$\begin{aligned} [\ *\ \ i = 1..n : A(i)\ ] &\ \widehat{=}\ \ A[1/i] * \ldots * A[n/i] \\ [\ \|\ \ i = 1..n : \mathcal{A}(i)\ ] &\ \widehat{=}\ \ \mathcal{A}[1/i] \parallel \ldots \parallel \mathcal{A}[n/i] \end{aligned}$$

Here the asterisk '$*$' denotes any action constructor mentioned above. Note that in general the index variable does not have to run through a range of values. In this case, the set of values is defined explicitly, for example: $i \in \{5, 2, 6, 9\}$. The leftmost value is considered first, the rightmost value last. The actions $A(i)$ are called *parameterized* actions, and, naturally, i is the *parameter*. The same rules are valid for the action systems in the second definition above.

3 Bus Components

A bus is a cost-effective data communication connection between two or more communicating devices. It is composed of *data*, *address* and *control* signals. The components of a concrete bus-based system include *masters*, *slaves* and an *arbiter* [1, 4, 12]. A master is a module that actively requests services from passive modules connected to the bus, the slaves. The master devices request the bus from the central control unit, the arbiter, which grants the bus to only one master at a time. This requires dedicated request and grant wires between each master and the arbiter.

In this section, we describe the variables and communication protocols that we use to model an asynchronous bus inspired by the AMBA AHB [1], supporting up to 16 bus masters. The interface between the bus components in our approach is shown in Figure 2. The actual Action System specifications of the components are presented later in section 4.

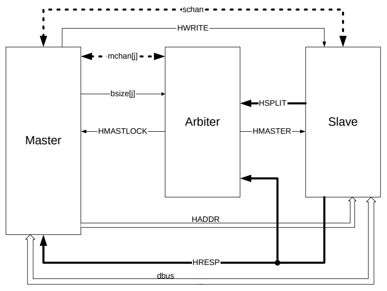

Fig. 2. System interconnections.

The asynchronous realization of the bus introduces more complex communication protocols than the synchronous one, mainly due to the fact that the participants to a given transfer have to know *when* data is actually ready to be either read or written. Hence, in asynchronous communication the active party sends a request signal to the passive party, which then responds by sending back an acknowledgement signal after completing the requested task.

Master-Arbiter communication. The asynchronous communication channel between the bus controller, i.e., the arbiter, and a master j, where $j = 0..15$ is a number identifying a particular master, is modeled by the 5-value variable *mchan*[j] of the type *mchanType* $\widehat{=}$ { *hr, hrl, gr, done, idle* }.

The initial value of *mchan*[j] is *idle*. The master requests the bus from the arbiter by setting *mchan*[j] to either *hr* (bus request) or *hrl* (request for a locked transfer). The arbiter responds eventually by setting the channel to *gr* (bus grant) and the control signal *HMASTLOCK* according to the request. When the master completes a transaction with a slave, it updates the signal *bsize*[j] (burst size, $bsize[j] = 0..16$), which carries the number of data words still to be transferred between the master and the slave, and sets *mchan*[j] to *done*. If $bsize[j] \neq 0$, the arbiter responds by setting the channel to either *hr*, *hrl*, or *gr* depending on the response received from the slave. If $bsize[j] = 0$, i.e., the whole burst has been transferred, the arbiter initializes *mchan*[j] to *idle*.

The values *hr*, *hrl*, and *gr* of *mchan*[j] correspond to the AMBA AHB signals *HBUSREQx*, *HLOCKx*, and *HGRANTx* , respectively [1].

Master-Slave communication. The master-slave communication channel is modeled by the variable *schan* of the type *schanType* $\widehat{=}$ {*req, ack* }. This is accompanied by the natural-type variables *dbus* and *HADDR* that model the data and address signals of the bus, the boolean variable *HWRITE* which specifies whether the master is reading (*HWRITE* = *false*) or writing (*HWRITE* = *true*) data, and the variable *HRESP* of the type *SlaveResponseType* $\widehat{=}$ {*OKAY, ERROR, RETRY, SPLIT* } through which the selected slave signals the status of the latest transaction. All the mentioned variables are globally shared by all the masters and slaves in the system.

The address *HADDR* of the AMBA AHB is composed of two parts: the N most significant bits represent the identification of the slave, while the remaining $32-N$ bits, assuming a 32-bit address, represent a valid location in the memory space of the selected slave. Hence, in our abstract model *HADDR* is a natural number with the range $0..2^{32}-1$, from which the identification number *SlaveId* of the selected slave is computed by: $SlaveId = HADDR/2^{32-N}$. The maximum number of slaves in the system is then 2^N.

Once the access to the bus is granted by the arbiter, the granted master first sets the *HWRITE* signal, the slave address *HADDR*, and also data *dbus* if *HWRITE* = *true* indicating a write operation, and places the channel *schan* to *req*. The selected slave tries to execute the requested operation and assigns then an appropriate value to *HRESP*. If *HRESP* = *OKAY* and *HWRITE* = *false* indicating a successful read operation, the slave also assigns a valid value to *dbus*. Then it acknowledges the master's request by setting *schan* to *ack*. In the case of a successful read, the master stores the value of *dbus* set by the slave. If *HRESP* = *OKAY* and *HMASTLOCK* = *true*, i.e., a successful locked transfer event (write or read) took place, the master initiates immediately a new communication cycle with the slave, as long as the burst has not yet been completed ($bsize[j] \neq 0$). If the slave sets *HRESP* to *ERROR* as the response to a locked transfer event, it is up to the master to decide whether to continue the locked burst with the slave or to interrupt it by communicating with the arbiter. In any other case the master always communicates first with the arbiter which then decides whether the master still has the access to the bus or not.

In the AMBA AHB specification, the slave may delay the transfer on the bus by making use of the *HREADY* signal. Basically, both the data and *HRESP* signal values are ignored, unless *HREADY* is high. Thus, the slave may insert *wait states* in the transaction flow. Due to the asynchronous communication protocol on the channel *schan*, the *HREADY* signal is not needed in our descriptions, as the master waits for the slave to set the channel *schan* to *ack* at every transfer event.

Another difference is that our abstract model does not contain an explicit decoder which generates the slave select signals *HSELx* from the address *HADDR* as in the AMBA AHB specification. Instead, a slave itself performs this selection based on the value of the address variable *HADDR* which is shared by all the slaves.

Arbiter-Slave communication. The interface between the arbiter and the slaves consists of the shared variables $HMASTER = 0\ ..15$, *HRESP* (defined above), and *HSPLIT*[0..15] (array of booleans).

The arbiter reports the master that is currently accessing the bus by assigning its identification number to the variable *HMASTER*. The arbiter reads the value of the variable *HRESP* which is set by a slave as a response to a transaction requested by the master accessing the bus. If *HRESP* = *SPLIT*, the slave has decided to split the current transfer, i.e., to postpone a part of the transfer burst. In this case, the arbiter masks or disables the request signal of the master identified by *HMASTER*. When the slave is later ready to continue the burst with a masked master j, it sets the boolean variable *HSPLIT*[j] to *true*. The arbiter then responds by removing the mask allowing the master j to access the bus again. If *HRESP* = *RETRY*, the slave wishes to retry the latest transaction with the master *HMASTER*. The arbiter gives the bus to this master immediately, provided there are no higher priority requests present. Otherwise the master has to compete with the other masters normally.

4 Specification

In this section we illustrate characteristics of the three components of a bus system, the arbiter, the slaves and the masters. The last two are abstract *templates* that we show in Figure 3 and Figure 4, respectively.

We consider a fixed number of masters, 16, and a generic number of slaves, specified as $NrSlaves = 2^N$, with *SlaveId* running from 0 to *NrSlaves* - 1. The

```
sys Slave (HADDR : natural 0..2^32; HMASTER : natural 0..15;
           dbus : natural; schan : SchanType;
           HSPLIT[16], HWRITE : bool; HRESP : SlaveResponseType) ::
|[
var  mem : natural; MastQ[16], busy : bool
const SlaveId := natural
actions S_1 : schan = req ∧ SlaveId = HADDR/2^(32-N) →
                (¬busy → HWRITE → mem := dbus
                   ▯ ¬HWRITE → dbus := mem
                 ▯ busy → skip);
                busy := b.b ∈ {true, false}
        S_2 : busy → HRESP := h.h ∈ {RETRY, SPLIT}
              ▯ ¬busy → HRESP := h.h ∈ {OKAY, ERROR}
        S_3 : HRESP = SPLIT → MastQ[HMASTER] := true;
        S_4 : S_1; S_2; S_3; schan := ack
        S_5 : (∃j.MastQ[j] → HSPLIT[j] := true; MastQ[j] := false)
                ▯ (⋀_{j=0}^{15} ¬MastQ[j] → skip)
init  MastQ, HWRITE, HSPLIT, busy := false; schan := ack;
      mem, HADDR, HMASTER, dbus := 0; HRESP := OKAY
do  S_4//S_5 od
]|
```

Fig. 3. The Asynchronous Slave Specification.

whole bus-based system is then represented by the parallel composition:

$$\begin{aligned}
&\mathcal{A}rbiter \\
\| \; &[\| \; j = 0..15 : \; \mathcal{M}aster(j)[mchan[j], bsize[j]/mchan, bsize] \\
\| \; &[\| \; i := 0..NrSlaves : \; \mathcal{S}lave(i)[i/SlaveId]]
\end{aligned}$$

In the composition, the $\mathcal{A}rbiter$ runs in parallel with the other system components, the masters and the slaves. We identify the master systems in the above description by the corresponding channel they share with the arbiter ($mchan[j]$ and $bsize[j]$). The $SlaveId$ number identifies the slaves.

Even though the arbiter that we analyze here is also a general template, we discuss it in more detail, as the generality only refers to the number of masters and slaves present in the system and not to the actual functionality of the arbiter, which is completely defined.

4.1 The Arbiter

The representation of the bus controller, modeled by the system $\mathcal{A}rbiter$, is illustrated in Figure 5.

Priorities. We leave the decision on the priority scheme to be decided at later steps in design. We only consider that the masters are organized in priority groups. If a master belongs to a higher priority group, it is allowed to interrupt the access to the bus of another master placed in a lower priority group. In the exemplification we give here, the masters belong to three priority groups. The groups are identified as the sets $G_k, \; k = 0..2$ declared in the **const** clause of the system $\mathcal{A}rbiter$. These sets contain the index by which the arbiter identifies the masters in the system. The group G_0 contains the masters having the highest priority.

```
sys $\mathcal{Master}$ $(mchan : MchanType; schan : SchanType; bsize : natural\ 0\ ..\ 16;$
        $HMASTLOCK, HWRITE : bool;\ HADDR : natural\ 0\ ..\ 2^{32}; dbus : natural;$
        $HRESP : SlaveResponseType) ::$
|[
var $mem, ssel, addr : natural;\ write : bool;$
actions $M_1 : mchan = idle \rightarrow$
            $(\quad bsize = 0 \rightarrow$
                $write, bsize, ssel :=$
                    $w, b, s.(w \in bool) \wedge (1 \le b \le 16) \wedge (0 \le s \le 2^N - 1)$
            $[\!]\ bsize \ne 0 \rightarrow skip);$
          $mchan := m.m \in \{hr, hrl\}$
        $M_2 : mchan = gr \rightarrow HWRITE, addr := write, a.0 \le a \le 2^{32-N} - 1;$
            $HADDR := 2^{32-N} \cdot ssel + addr;$
            $(HWRITE \rightarrow dbus := db.db \in natural\ [\!]\ \neg HWRITE \rightarrow skip);$
            $schan := req$
        $M_3 : schan = ack \rightarrow$
          $(\quad HRESP = OKAY \rightarrow$
                $(\neg HWRITE \rightarrow mem := dbus\ [\!]\ HWRITE \rightarrow skip); bsize := bsize - 1$
            $[\!]\ HRESP = ERROR \rightarrow bsize := b.(b = bsize - 1) \vee (b = 0)$
            $[\!]\ HRESP \in \{SPLIT, RETRY\} \rightarrow skip);$
        $M_4 : (\neg HMASTLOCK \vee (bsize = 0) \vee (HRESP \in \{SPLIT, RETRY\}) \rightarrow$
              $mchan[j] := done$
          $[\!]\ HMASTLOCK \wedge (bsize \ne 0) \wedge (HRESP \notin \{SPLIT, RETRY\}) \rightarrow$
              $skip)$
        $M_5 : M_3; M_4$
 init $bsize, mem, ssel, addr, HADDR, dbus := 0; mchan := idle; schan = ack;$
      $write, HMASTLOCK, HWRITE := false; HRESP := OKAY$
 do $M_1\ [\!]\ (M_2\ ;\ M_5)$ od
]|
```

Fig. 4. The Asynchronous Master Specification.

```
sys $\mathcal{Arbiter}\,(mchan[16] : MchanType; HMASTER : natural\ 0 \ldots 15;$
    $HMASTLOCK, HSPLIT[16] : bool; bsize : natural\ 0 \ldots 16;$
    $HRESP : SlaveResponseType) ::$
|[
var       $mask[16], ret[16], reserved : bool$
const  $G_0 := \{0, 1, 2, 3, 4\}; G_1 := \{5, 6, 7, 8, 9, 10, 11\}; G_2 := \{12, 13, 14, 15\}$
expressions  $NoRetries(j, k) \mathrel{\widehat{=}} (\forall i \in G_k \,.\, i \ne j \Rightarrow \neg ret[i])$
actions  $Grant(j, k) :$     $(mchan[j] \in \{hr, hrl\}) \wedge (mask[j] \vee HSPLIT[j])$
                             $\wedge\ NoRetries(j, k) \wedge \neg reserved \rightarrow$
                                 $HMASTLOCK, HMASTER := (mchan[j] = hrl), j;$
                                 $mask[j], reserved := true; ret[j] := false;$
                                 $mchan[j] := gr$
         $ChkResp(j) :$        $mchan[j] = done \rightarrow$
                                 $(Split(j)\ [\!]\ Retry(j)\ [\!]\ Ok\_or\_Err(j));$
                                 $reserved := false$
         $Split(j) :$          $HRESP = SPLIT \rightarrow mask[j] := false;\ Req(j)$
         $Retry(j) :$          $HRESP = RETRY \rightarrow ret[j] := true;\ Req(j)$
         $Ok\_or\_Err(j) :$    $HRESP \in \{OKAY, ERROR\} \rightarrow$
                                 $bsize[j] \ne 0 \rightarrow Req(j)\ [\!]\ bsize[j] = 0 \rightarrow mchan[j] := idle$
         $Req(j) :$              $\neg HMASTLOCK \rightarrow mchan[j] := hr$
                             $[\!]\ HMASTLOCK \rightarrow mchan[j] := hrl$
 init  $mask := true; HSPLIT, ret, reserved, HMASTLOCK := false;$
       $mchan := idle; HRESP := OKAY; bsize, HMASTER := 0$
 do  $[//k = 0..2 : [\ [\!]\ j \in G_k : (Grant(j, k)\ [\!]\ ChkResp(j))]]$ od
]|
```

Fig. 5. The Asynchronous Arbiter.

Actions composing the $\mathcal{A}rbiter$ system. In the following, we analyze the behavior of the system $\mathcal{A}rbiter$, by describing the behavior of its actions.

In the **actions** clause of the system we specified six actions. The notation also provides a generic aspect, illustrated by the action parameters k and j. We describe the behavior of the controller with respect to a single master, and the parameters help generalizing, so that we cover all the other masters. In the following we refer to the system's actions without mentioning the parameters.

Action Grant. This action is responsible for granting the bus ownership to a master requesting it ($mchan[j] \in \{ hr, hrl \}$), while: the module is not masked, or it received an *HSPLIT* update ($mask[j] \vee HSPLIT[j]$), the bus is not reserved, and the answer from the last operating slave was not *RETRY*. Consequently, the bus is granted, not before updating the system with the right information concerning the current master, the mask value, etc.

Action ChkResp. The second important action of the arbiter models the answer of the controller following the different possible results communicated by the operating slaves. The behavior is constructed on the included actions, *Split*, *Retry*, *Ok_or_Error* and *Req*. The communication variable $mchan[j]$ is updated correspondingly in each situation characterized by a given *HRESP* value.

Action Split. The slave may decide not to answer immediately to the request coming from the current owner of the bus. We modeled this by assigning an arbitrary value to the slave's local variable *busy* (Figure 3). If the slave issues a *SPLIT* answer to the request of the master, the arbiter then inhibits the master from taking part in the arbitration process by setting the corresponding mask element to *false*. The value of every mask element is checked in the action *Grant*. Whenever the slave decides to allow the master to (re)start the operations, it informs the arbiter by setting the corresponding element of the *HSPLIT* vector to *true*. This is also checked by the action *Grant*. If the slave is ready to resume a previously split connection, then the mask is updated next time the specific master obtains the access to the bus.

Action Retry. Similar to the above action *Split*, this action, by assigning $ret[j] :=$ *true*, does not allow any other master from the same priority group with the current owner of the bus to gain access. If there is no higher priority master requesting, the ownership of the bus will remain to the current master.

Action Ok_or_Error. From the point of view of the arbiter, an *OKAY* or an *ERROR* answer from the operating slave bears the same significance. The decision in an erroneous situation is taken by the master (action M_3, Figure 4). The arbiter only checks the value of the corresponding *bsize* element and either puts the specific channel on *idle*, or continues in a normal manner, consequently.

Global behavior. The behavior of the arbiter is described further by the composition $[//k = 0..2 : [\ \llbracket\ j \in G_k : (Grant(j,k)\ \llbracket\ ChkResp(j))]]$. The actions within the innermost quantified composition deal with masters coming from the same priority group. The outermost quantified composition describes the priority scheme.

5 Discussion

Bus Access. In the original AMBA bus specification [1] the bus is governed by a global clock. All the transfers relate to this signal which synchronizes the data transfers and the control lines. In the previous sections we described an asynchronous approach to bus control. Instead of the clock signal we introduced communication channels that identify the moments when data or control signals are valid on certain lines.

The transfers allowed on the synchronous bus are not only limited in size, but also, based on the clock frequency, in time. Thus, a sixteen-word transfer can be realized in sixteen clock cycles if the master that controls it receives no interruption. Knowing the clock frequency, one can establish a period of time in which a normal executed transfer takes place. In the asynchronous representation, even though the transfer size is also limited, one cannot be certain of the time period in which a sixteen-word transfer is completed, in an uninterrupted execution.

Even though, at this level of the description, the period of *time* in which a certain master controls the bus was ignored, we can also think of means by which one can control this aspect of the arbitration. For instance, a local synchronous counter can be attached to the arbiter so that it monitors, in steps that can be related to the actual time of the processing, the actual time a master has been accessing the bus. Another solution could be the employment of a specific master, which would implement the same counter. This master would have the highest priority in the system, and thus, could interrupt the access to the bus of any other master. By not addressing any slave, the interrupt requests sent by this master cannot be masked by a possible split or retry operation.

Refinement issues. The abstract bus specification, given above as an action system description, is intended to be developed into a hardware-realizable model in a stepwise manner using Refinement Calculus-based [2] *transformation rules*. Such a disciplined design flow yields a concrete system model which is a logically correct implementation of the initial abstract specification, satisfying possibly a set of auxiliary logical constraints which are necessary for successful circuit implementation [8, 10]. This lays a solid ground for the technology mapping process, where the final formal description is transformed into a layout of actual circuit components.

After the initial specification, the abstract bus arbiter and the involved masters and slaves are further decomposed or partitioned into compositions of action systems each of which describes some essential functional aspect of an original system component. Each decomposition step is a refinement, where a new communication channel is created between the separated parties. The majority of the introduced channels are local to the system components, which means that they do not change the bus itself. However, some transformations may involve the bus as well. For example, if the address decoding operation is extracted from the slaves, a new global control block is created for the bus. Splitting the variable *dbus*, which models the data bus between the masters and slaves, into two separate variables modeling write and read buses is an example of another kind of global refinement.

The decomposition procedure is followed by the *handshake expansion*, a non-trivial *data refinement* [2], where the description is brought closer to the circuit level by implementing each abstract communication channel with a set of boolean variables.

As an example, let us consider a master-arbiter channel *mchan* (we have omitted the index j to make the notation simpler) which has 5 possible values $\{hr, hrl, gr, done, idle\}$. We can implement it using 5 boolean handshake variables or signals, say *hr*, *hrl*, *gr*, *done*, and *next*. They are related to the original variable by the following relations:

$$\begin{array}{l}
R_1 \mathrel{\widehat{=}} (mchan = hr) \Leftrightarrow (hr \wedge \neg hrl \wedge \neg gr \wedge (\neg done \vee next)) \\
R_2 \mathrel{\widehat{=}} (mchan = hrl) \Leftrightarrow (\neg hr \wedge hrl \wedge \neg gr \wedge (\neg done \vee next)) \\
R_3 \mathrel{\widehat{=}} (mchan = gr) \Leftrightarrow ((hr \vee hrl) \wedge gr \wedge (\neg done \vee next)) \\
R_4 \mathrel{\widehat{=}} (mchan = done) \Leftrightarrow \\
\qquad\qquad ((((hr \vee hrl) \wedge done) \vee (\neg hr \wedge \neg hrl \wedge \neg done)) \wedge gr \wedge \neg next) \\
R_5 \mathrel{\widehat{=}} (mchan = idle) \Leftrightarrow (\neg hr \wedge \neg hrl \wedge \neg gr \wedge \neg done \wedge \neg next)
\end{array}$$

Hence, the resulting communication channel is considered idle when all of its handshake variables are *false* (R_5). The channel is put to a request state, when the master sets either *hr* or *hrl* to *true*, or when the arbiter decides to remove grant by setting the grant signal *gr* to *false* and asserting the *next* signal (R_1, R_2). The arbiter puts the channel to the grant state by setting *gr* to *true* as the response to the asserted request *hr* or *hrl*, or by setting the variable *next* to *true* as the response to the asserted *done* signal while keeping *gr true* (R_3). The *done* state indicates that the master has set either the variable *done* to *true*, or the request *hr* or *hrl* to *false* (R_4). In the former case, the master has just transmitted or received a data word, but the burst has not yet been completed. In the latter case, the last data of the burst has just been transmitted or received, which means that the arbiter can put the channel to the initial *idle* state.

In Figure 6, we illustrate the correspondence between the abstract channel and the refined, concrete one. However, the figure is a simplified representation, as it does not include the *hrl* value of the abstract channel.

The design flow continues, after the transformation of the communication channels, with the *circuit extraction*, where the formal description of each system component is stepwise refined into a concrete description from which an actual circuit implementation can be easily derived. The final phase in the Action Systems-based design flow is the *implementation* or *technology mapping* step, where the system units are refined into compositions of action systems each of which corresponds to either an actual circuit component in a component library available to the designer or a synthesizable register-transfer-level hardware description language (HDL) model.

Observe that the design flow becomes more fluent, if our component library contains abstract action system models for large circuit components that have been verified formally beforehand. For example, we could have an entire bus master or slave, or a significant part of it, as a single library component. Furthermore, once the shared bus controller (arbiter) has been designed from an abstract specification to the circuit implementation, it is placed in a library and re-used from there. Then the design flow of a system built around the bus discussed in this paper would mainly consist of the decomposition phase, where the abstract models of the pre-defined library components are stepwise extracted from the initial system specification.

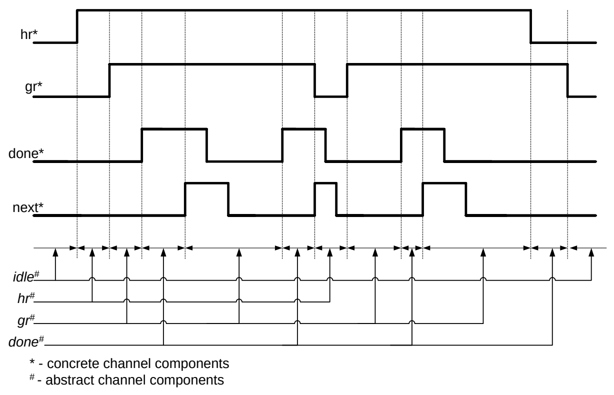
hr*
gr*
done*
next*
idle#
hr#
gr#
done#
* - concrete channel components
# - abstract channel components

Fig. 6. The transformation of the channel *mchan*.

6 Conclusions

We presented in this study a formal representation of an asynchronous bus structure. It originally started with the intention to describe an asynchronous AMBA bus, which, in comparison with similar approaches to bus modeling [4, 12], seemed to be a less complex structure. Due to the specifics of the asynchronous architecture the descriptions, the modeling eliminated several characteristics of the original AMBA structure. Some of the common aspects are the centralized arbitration, the split transfer features and the selection of the slave. Some details of the AMBA specification are abstracted away in our presentation, due to the higher level of description. This is the case with the data bus, the communication channels and aspects of the slave selection.

One of the purposes of this study was to integrate the analyzed bus structure into our formal framework for digital hardware design as a means of allowing different devices to communicate and exchange data. By representing the bus in the same framework with the other systems in a design process, we have the possibility to apply the same techniques on both the systems modeling the digital hardware devices and the systems describing the underlying connectivity. The (successive) transformations one needs to perform in order to bring the descriptions of these systems to more concrete levels are executed in the formal framework of Action Systems, thus ensuring a correct derivation process. As an example, we described the transformations applied to the communication channel *mchan* in order to bring its representation at a lower level, where physical interconnection lines can be identified in the description. We can apply the same technique in order to bring the representation of the slave channel *schan* closer to hardware implementation levels.

We provided templates for the components to be included in bus-oriented design architectures. There is no specification on the internal architecture of these devices, as they may be either synchronous or asynchronous implementations.

There are several other aspects of bus design that are subject to subsequent studies. For instance, we did not specify, yet, how the arbiter may terminate the ownership of the bus in case of a prolonged access, although we offered some possible scenarios. An immediate follower of this work can start with the analysis of the other possibilities offered by the original AMBA specification, the System Bus (ASB) and the Peripheral Bus (APB).

References

1. ARM Limited. AMBA Specification (Rev 2.0), 1999.
2. R. J. R. Back and J. von Wright. *Refinement calculus: A Systematic Introduction.* Springer. April 1998.
3. W.J.Bainbridge. *Asynchronous System-on-Chip Interconnect.* PhD. Thesis, University of Manchester, UK, 2000.
4. W.J.Bainbridge and S.B.Furber. Asynchronous Macrocell Interconnect Using MARBLE. In Proceedings of the 4^{th} International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '98) San Diego, CA, March 30 - April 2, 1998.
5. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.
6. J. Hooman. *Verifying Part of the ACCESS.bus Protocol using PVS.* Proceedings 15th Conference on the Foundations of Software Technology and Theoretical Computer Science, LNCS 1026, Springer-Verlag, pages 96-110, 1995.
7. A. Mokkedem, R. Hosabettu, M. D. Jones, G. Gopalakrishnan, *Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem.* Formal Methods in Systems Design, vol. 16, no. 1, pp. 93-119, January 2000.
8. J. Plosila. *Self-Timed Circuit Design - The Action Systems Approach.* Ph.D. Thesis, University of Turku, Dept of Applied Physics, Turku, Finland, 1999.
9. C. Purtell-Tappen. *Platform Express to Accelerate Platform-Based System-on-Chip Design and Verification.* ECN Magazine, September 2001.
10. T. Seceleanu. *Systematic Design of Synchronous Digital Circuits.* Ph.D. Thesis, Abo Akademi, Turku, Finland, 2001.
11. T. Seceleanu, J. Plosila. *Hierarchical Action Systems.* Manuscript. To appear as Technical Report.
12. A. Zitouni et All. *Design of an Asynchronous VME bus Controller for Heterogeneous systems.* Dedicated Systems Magazine - 2000 Q3 (http://www.dedicated-systems.com).