ISSN 1292-862

TIMA Lab. Research Reports

A Functional Approach to the Formal Specification of Networks on Chip

J. SCHMALTZ*, D. BORRIONE*

* TIMA Laboratory, 46 avenue Félix Viallet 38000 Grenoble France

ISRN TIMA--RR-04/04-01--FR

TIMA Laboratory,
46 avenue Félix Viallet, 38000 Grenoble France

A Functional Approach to the Formal Specification of Networks on Chip

Julien Schmaltz[1] and Dominique Borrione

TIMA Laboratory, VDS Group
Joseph Fourier University
46 avenue Felix Viallet
38031 Grenoble Cedex, France
{Julien.Schmaltz, Dominique.Borrione}@imag.fr

Abstract. We present a functional approach, based on the ACL2 logic, for the specification of *system on a chip* communication architectures. The functional style allows the method to be compositional for both system definition and validation. When performed in the context of the ACL2 logic, all the definitions and theorems are not only reusable, but also constitute an executable and proven valid specification for the system. We illustrate the approach on a state of the art network on chip: the *Octagon*. We prove that messages travel over this network without being modified and eventually reach their expected destination.

1 Introduction

The design of a production quality *system on a chip* (SoC), involving several processor-like modules, and several memories and peripherals is well supported by a variety of simulation and formal verification tools at the register transfer level (RTL) and below, where ninety percent of the design time is spent [1]. In contrast, the initial phase when concepts are first written down, and which involves critical decisions on the number of units, their communication, the main pipelines, memory size and global system performances, is only supported by simulation tools taking as input relatively *ad hoc* formalisms. It is far from obvious that the RTL implementation is ever checked compliant with the description model produced at the concept phase. Yet, as systems increasingly reuse pre-existing processor and memory cores, which have been verified intensively in isolation, an essential aspect of an overall system functional correctness relies on the correctness of their communications. In this context, the work reported here focuses on the specification of the communications on a chip, specified at a high-level of abstraction, and involving generic network components where the number of interconnected modules is finite, but possibly not fixed. Our objective is to develop a high-level specification and verification method for the concept phase,

[1] Part of this work was done while visiting the Department of Computer Sciences of the University of Texas at Austin. This visit was supported by an EURODOC scholarship granted by the "Region Rhone-Alpes", France.

that will support validating that the intended communication module is correct. Simulation, semi-formal and formal validation techniques are all considered complementary, and therefore targeted by our study. Logic level automatic verifiers, such as property and model checkers, are not directly applicable, due to their lack of abstract data types, and the fact that they apply on fixed, rather than generic, structures. Symbolic simulation and theorem proving, on the contrary, provide reasoning support to prove properties such as: a token travels correctly, and arrives to its destination on a *parameterized* communication structure.

In this paper, we present the formal specification, in the ACL2 logic [2], of a state of the art network on chip developed by *ST Microelectronics* and named *Octagon*. Our purpose is the translation of the informal descriptions presented in the scientific literature ([3] and [4]), into a formal description in a computational logic, where the formal model is both executable and can be validated by the proof of theorems. Our modeling style is functional, *i.e.* components and their interconnections are represented by sets of functions. Our approach is *compositional* for both the modeling and the validation process, which eases the elaboration of models and proofs. Our contribution is a specification and functional verification methodology for the very first design steps of parameterized communication virtual modules. To our knowledge, the application of automatic theorem proving to on-chip communication hardware is new. To this end, we have developed a specific library of functions and theorems, most of which will be reusable for circuits of similar functionality.

In the next section, we introduce the functional formalism used for the specification of *Octagon*. We present the functional definition of the transfer of messages and the general theorems that express its correctness. Section three introduces the main features of the *Octagon* and of the interconnected nodes. In section four, the overall memory structure and the functional specification of a node system are presented. Within a node system, we prove that the local communications are correct according to the formalism of section two. The functional definition of the *Octagon*, and the main theorems that validate this model are given in section five. We prove that messages eventually reach their expected destination without being modified. In the final section, we discuss the experimental results, and present our conclusions.

2 Functional Specification of Communication Systems

2.1 Principles of communications

We consider communications between a master - typically a processor (DSP, MCU...) - and a slave - typically a co-processor or a memory (ASIC, DMA...). In this configuration, a master starts the communication by sending an *order* to the slave, which replies with a *result*. An *order* is a list of three elements: an *operation*, a *location* and an *item*. A *result* is a list of two elements: the status *Rstat* of the operation and the returned item *Ritem*. For the purpose of this paper, we assume masters to be processors and slaves to be memories. Consequently, an *operation* is a *read* or *write* to a given location, or *no_op* to indicate

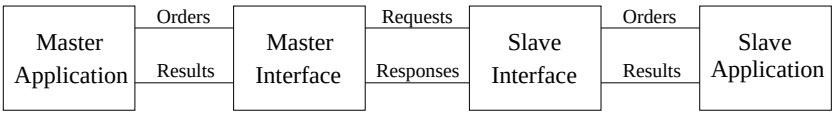

Fig. 1. Point to point interconnection scheme

an idle state of the master. *Rstat* takes the value *OK* if the communication is successful. For a memory, it can be *inv_addr* to indicate a wrong address or *inv_op* to indicate an unknown operation. Communication operations are *orthogonal* to the computation operations [5]. They are separated in two classes of components: the interfaces and the applications (Figure 1). To distinguish between interface-application and interface-interface communications, the former dialogue is denoted by *orders* and *results*, the latter by *requests* and *responses*. Generally, the encoding of orders differs from the encoding of requests.

A *request* has the same structure as an order; it is a list of three elements: request = (r/w addr data), where r/w = r when operation is a read and r/w = w when operation is a write. Parameter *data* is meaningful for a write request, in the case of a read it takes value '-' to denote a "don't care". A *response*, like a *result*, is a list of two elements: response = (status Rdata). *status* corresponds to *Rstat*, and takes the same possible values: *OK*, *inv_addr*, *inv_op* ....

We distinguish data a and b from their memory addresses $@a$ and $@b$. Let us consider the two communication events below:

1. (read @a -)
2. (write @a "Hello")

Communication event 1. The master interface receives the read order and produces the request *(r @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.
Communication event 2. The master interface receives the write order and produces the request *(w @a "Hello")*. The slave interface receives this request and computes the order *(write @a "Hello")*. The slave application writes the string "Hello" at the address $@a$ and the result *(OK "Hello")* is sent back to the master application.

In the following, we use the element name for the function that returns an element of a communication event. For instance, *Operation*, *Location*, *Item* are the functions that return the first, second and third element of an *order*.

2.2 Functional Modeling of Communications

An interface communicates with two components, an application and another interface, and is thus modeled by two functions. For the master interface: MI_{req} computes a *request* from an *order*; MI_{res} computes a *result* from a *response*. For the slave interface: SI_{ord} computes an *order* from a *request*; SI_{resp} computes a *response* from a *result*. The transfer of an order from the master to the slave application is defined as the composition of MI_{req} with SI_{ord}:

Definition 1. *Transfer of an order*

$$Trans\_ord(order)\ returns\ Order \stackrel{def}{=} SI_{ord} \circ MI_{req}(order)$$

A transfer from the slave to the master application is defined as the composition of MI_{res} with SI_{resp}:

Definition 2. *Transfer of a result*

$$Trans\_res(result)\ returns\ Result \stackrel{def}{=} MI_{res} \circ SI_{resp}(result)$$

Let function *Slave* model the slave application; a complete transfer between the master and the slave is defined by the composition of $Trans\_res$, $Slave$ and $Trans\_ord$:

Definition 3. *Transfer*

$$Transfer(order)\ returns\ Result \stackrel{def}{=} Trans\_res \circ Slave \circ Trans\_ord(order)$$

The actual definitions are slightly different. The slave interface functions take an additional input: a flag *select*. If *select* is *true* then the interface is active and each function computes an order or a response. Otherwise, the functions return error values meaning that they are not ready to operate. Consequently, the composition of these "concrete" functions is not exactly a mathematical composition. Let us consider the functions MI_{req} and SI_{ord}:

Definition 4. *Master Interface*

$MI_{req}(order)\ returns\ Request \stackrel{def}{=}$
if $Operation(order) = read$ **then** *; read request*
$list(r, Location(order), Item(order))$
else *; write request*
$list(w, Location(order), Item(order))$
endif

Definition 5. *Slave Interface*

$SI_{ord}(request, select)\ returns\ Order \stackrel{def}{=}$
if *select* **then** *; slave interface is active*
if $r/w(request) = r$ **then** *; read operation*
$list(read, Addr(request), Data(request))$
else *; write operation*
$list(write, Addr(request), Data(request))$
endif

else ; *slave is not active*
$list(not\_ready, no\_loc, no\_item)$
endif

The transmission of an order is therefore defined by:

Definition 6. *Transmission of an Order*
$trans\_ord(order, select)$ *returns Order* $\stackrel{def}{=} SI_{ord}(MI_{req}(order), select)$

The correctness of the transmission of an order is achieved if the order received by the slave application is "equal" to the order sent by the master application. Generally, the function *Addr* used in the SI_{ord} function will modify the *location* of the original order to satisfy a specific mapping of the slave application addresses. Consequently, the received order by the slave application is not strictly equal to the sent order, but equal *modulo* a given address mapping.

Theorem 1. *Trans_Ord Correctness*
$\forall\, order,\ trans\_ord(order,\ true) \simeq order$

Proof. The proof follows obviously, by rewriting, from the definitions of MI_{req} and MSI_{ord}, which imply the following propositions (1), (2), (4). Proposition (3) depends on the definitions of *Addr* and $\simeq_{map}$. In the most simple case, *Addr* performs no computation, and $\simeq_{map}$ is equality.

$$Operation(order) = read \implies r/w(request) = r \quad (1)$$
$$Operation(order) = write \implies r/w(request) = w \quad (2)$$
$$Location(order) \simeq_{map} Addr(request) \quad (3)$$
$$Item(order) = Data(request) \quad (4)$$

□

Following a similar approach, the transmission of a result is:

Definition 7. *Transmission of a Result*
$Trans\_res(result, select)$ *returns Result* $\stackrel{def}{=} MI_{res}(SI_{resp}(result), select)$

The correctness of the transmission of a result is achieved if the result received by the master application is equal (generally strictly) to the result sent by the slave application. (The proof of the following theorem would be similar to the proof of Theorem 1).

Theorem 2. *Trans_Res Correctness*
$\forall\, result,\ Trans\_res(result,\ true) = result$

The definition of a transfer (definition 3) becomes:

Definition 8. *Transfer*
$Transfer(order)$ *returns Result* $\stackrel{def}{=}$
$Trans\_res(Slave(Trans\_ord(order,\ select)),\ select)$

The correctness of a transfer is achieved if its result is equal (again *modulo* an address mapping) to the application of the function *Slave* to the order produced by the master application.

Theorem 3. *Transfer Correctness*
$\forall\, order,\ Transfer(order,\ true)\ \simeq\ Slave(order)$

Proof. follows from theorems 1 and 2. □

In practice, master and slave interfaces are not directly connected. A communication medium, bus or network, determines how requests and responses are transferred between them. The medium architecture is modeled by a function *CommArch* which takes and returns a *response* or a *request*, *i.e.* a *message*. This function is inserted in the definitions 1, 2, 6 and 7 and only those. For instance, definition 1 becomes:

Definition 9. *Transfer of an order via a medium*
$Trans\_ord(order)\ returns\ Order\ \stackrel{def}{=}\ SI_{ord}\ \circ\ CommArch\ \circ\ MI_{req}(order)$

This modification is expressed in definition 6 as follows:

Definition 10. *Transmission of an Order via a medium*
$trans\_ord(order, select)\ returns\ Order\ \stackrel{def}{=}$
$SI_{ord}(CommArch(MI_{req}(order)), select)$

Two essential properties must hold on the *CommArch* function: the message is conveyed between the correct source and destination, and the message is unchanged: $CommArch(message)\ =\ message$. Under those conditions, Theorems 1, 2 and 3 remain valid after the introduction of the communication architecture. The remainder of this paper concentrates on the modeling and proof of these properties on the Octagon architecture.

3 Overview of the Octagon

3.1 Architecture and Routing

A basic Octagon unit consists in eight nodes and twelve bidirectional links (Figure 2). It has two main properties: two-hop communication between any pair of nodes and simple, shortest-path routing algorithm ([3] and [4]).

An *Octagon packet* is data that must be carried from the source node to the destination node as a result of a communication request by the source node. A scheduler allocates the entire path between the source and destination nodes of a communicating node pair. Non-overlapping communication paths can occur concurrently, permitting spatial reuse.

The routing of a packet is accomplished as follows. Each node compares the tag ($Packet\_addr$) to its own address ($Node\_addr$) to determine the next action. The node computes the relative address of a packet as:

$$Rel\_addr\ =\ (Packet\_addr\ -\ Node\_addr)\ mod\ 8 \tag{5}$$

At each node, the route of packets is a function of Rel_addr as follows:

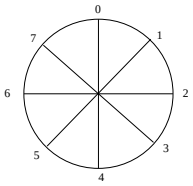

Fig. 2. Basic Octagon Unit

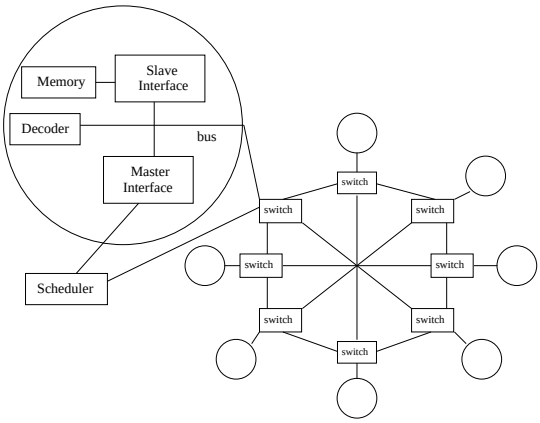

Fig. 3. Node System connected to a switch and the scheduler

- $Rel\_addr = 0$, process at node
- $Rel\_addr = 1$ or 2, route clockwise
- $Rel\_addr = 6$ or 7, route counterclockwise
- route across otherwise

Example 1. Consider a packet *Pack* at node 2 sent to node 5. First, $5-2 \; mod \; 8 = 3$, *Pack* is routed across to 6. Then, $5 - 6 \; mod \; 8 = 7$, *Pack* is routed counterclockwise to 5. Finally, $5 - 5 \; mod \; 8 = 0$, *Pack* has reached its final destination.

3.2 Node and System Structure

Each node of the network is a small system built around a bus architecture. It contains an address decoder, a master and a slave interface and a memory unit. This system is connected to *Octagon via* a switch (Figure 3). Master interfaces and switches are connected to the scheduler.

Nodes and Octagon work as follows. If the memory unit and the master interface involved in a communication belong to the same node, the request is said to be *local* and the output flag of the decoder is set to *true*. It is said to be *non-local* otherwise, and the output flag of the decoder is set to *false*. Suppose a non-local transaction is pending at several master interfaces. The scheduler determines the set of transactions that can be done concurrently, *i.e.* those that do not share communication paths. A path is allocated for each one of them and carries both the request and the response. When every concurrent transaction is finished, the system is ready for a new set of transactions.

In the next two sections, we present the functional specification and validation of the network, in algorithmic notation for readability. The actual model is written in LISP and can be found in [6] which also contains the ACL2 theorems. All proofs have been performed mechanically with the ACL2 theorem

prover, and the reader interested in the detailed proof steps can run the certified "book" available on the web [7].

4 Functional Specification of the Node System

4.1 Memory Structure

The overall system memory is equally distributed over the nodes. Let *ms* be the size of a memory in a node, and *Num_Node* be the number of nodes (8 for the Octagon, but the argument is more general). The size of the global memory is $global\_ms = Num\_Node \times ms$.

During transfers, the master uses the global address, which ranges from 0 to $global\_ms - 1$ and the slave selected by the decoder reads or writes the data to its local address *local_addr*. The local and global addresses are related by: $local\_addr = global\_addr \bmod ms$.

Conversely, the destination node possessing a given global address *global_addr* is the node i, such that

$$i = global\_addr \; div \; ms \tag{6}$$

From the above, a definition of relation $\simeq_{map}$ that preserves the validity of Theorem 1 is given by:

Definition 11. *Definition of* $\simeq_{map}$

$$x \simeq_{map} y \stackrel{def}{=} \begin{cases} \textbf{true} \; if \; y = x \bmod ms \\ \textbf{false} \; otherwise \end{cases}$$

4.2 Functional Memory Model

The memory unit of a node is modeled by a list *memo* of items and a function *Memory* that operates on *memo*. The address of an item is its position in the list.

Definition 12. *Memory*

$Memory(order,\ memo)\ returns\ (Status\ Item\ Memo) \stackrel{def}{=}$
  if $Location(order) < Len(memo)$ **then** *; valid address?*
    if $Operation(order) = read$ **then**
      $list(OK, Memo[Location(order)], memo)$
    elsif $Operation(order) = write$ **then**
      $list(OK, Item(order), Update(memo, Location(order), Item(order)))$
    else *; invalid operation*
      $list(inv\_op, no\_item, Memo)$
    endif
  else *; invalid address*
    $list(inv\_addr, no\_data, Memo)$
  endif
where L [i] denotes the i'th element of L
and Update(L, a, v) returns a list where the element at position a is v

The global memory $Glob\_Mem$ is represented by the ordered concatenation of all local memory lists *Memo*, starting from 0.

$$Glob\_Mem = (d_0\ d_1\ ...\ d_{ms-1}\ d_{ms}\ d_{ms+1}\ ...\ d_{num\_node \times ms-1}) \tag{7}$$

Two functions are defined on *Glob_Mem*: *get_local_mem* extracts the memory unit number *node_nb* from the global memory, and *update_local_mem* returns a global memory containing an updated local memory.

Definition 13. *Get Local Memory*

$get\_local\_mem(Glob\_Mem, node\_nb, ms)\ returns\ Memo \stackrel{def}{=}$
$first(ms, nthcdr(node\_nb \times ms, Glob\_Mem))$
where first(n,L) returns the n first elements of L and
nthcdr(n, L) returns L without its first n elements

Example 2. Consider *Glob_Mem* = (a b c d e f), ms=2 and Num_Node=3. The memory of node 1 is $first(2, nthcdr(1 \times 2, Glob\_Mem)) = first(2, (c\ d\ e\ f)) = (c\ d\ )$.

Definition 14. *Update Local Mem*

$update\_local\_mem(Glob\_Mem, memo, node\_nb, ms)\ returns\ Glob\_Mem \stackrel{def}{=}$
$concatenate(first(node\_nb \times ms, Glob\_Mem), memo,$
$nthcdr((node\_nb + 1) \times ms, Glob\_Mem))$

Example 3. The memory of node 1, as of example 2, is updated as follows:
$update\_local\_mem((a\ b\ c\ d\ e\ f), (g\ h), 1, 2) = concatenate((a\ b), (g\ h), (e\ f)) = (a\ b\ g\ h\ e\ f)$

4.3 Specification of the Node System

We define a function *Node* which represents a generic node system. Its execution models either one local communication or a step in a distant communication. Priority is given to the communication started by the local master. It takes three architectural parameters: *Glob_Mem*, *ms*, and the own node number *node_nb*. The other arguments are the pending order of the local master *ord*, the request *req* and the response *resp* coming from a distant node, and two Boolean flags stating the validity of these last two arguments. *Node* returns a list composed of the result of a local communication (the string "no_result" if none), the emitted request ("no_message" if none), the response to the incoming request ("no_response" if none), and the new value of the global memory.

Definition 15. *Node System*

$Node(ord, req, resp, Glob\_Mem, ms, validRequest, validResponse, node\_nb)$
$returns\ (Result\ Request\ Response\ Glob\_Mem) \stackrel{def}{=}$
if $Operation(ord) = no\_op$ **then**
if *validRequest* **then**
$(\text{“}no\_result\text{”}\ \text{“}no\_message\text{”}\ Netw\_transfer(req, Glob\_Mem, ms, node\_nb))$

elsif *validResponse* **then**
$(MI_{res}(resp)$ "*no_message*" "*no_response*" $Glob\_Mem)$
else
("*no_result*" "*no_message*" "*no_response*" $Glob\_Mem)$
endif
elsif $Decoder(Location(ord), ms)$ **then**
$(Bus\_transfer(ord, true)$ "*no_message*" "*no_response*" $Glob\_Mem)$
else
("*no_result*" $MI_{req}(ord)$ "*no_response*" $Glob\_Mem)$
endif

Local communications are represented by function $Bus\_transfer$ which is defined similarly to definitions 9 and 10 of section 2. Function *CommArch* is replaced by function *Bus*, which is here modeled by the identity function: $Bus(x) = x$. Consequently, the correctness of local operations follows from Theorems 1, 2, and 3.

At the beginning of a distant communication initiated by the master of node *nb_1*, the local order is *read* or *write* and function *Node* with parameter *node_nb* = *nb_1* calls MI_{req}. This produces a request which is sent over the network. At the destination node *nb_2*, the *validRequest* is set to "true" by the scheduler. This is modeled to a second call to function *Node* with *node_nb* = *nb_2* that calls function *Netw_transfer* below to compute the response. The response is sent back to the source node *nb_1*, with a third call to *Node* with parameters *node_nb* = *nb_1* and *validResponse* = *'true"*, that invokes function MI_{res} to compute the final result of the distant communication.

Definition 16. *Network Transfer*
$Netw\_transfer(req, Glob\_Mem, select, node\_nb)$
$returns\ (Response\ Glob\_Mem) \stackrel{def}{=}$
let ord **be** $SI_{ord}(req, select, ms)$ **in**
let $(res\ mem)$ **be**
$Memory(ord, get\_local\_mem(Glob\_Mem, node\_nb, ms))$ **in**
$list(SI_{resp}(res, select),$
$update\_local\_mem(Glob\_Mem, mem, node\_nb, ms))$

Distant communications are completed by function *Octagon*, presented in the next section.

5 Functional Specification of Octagon

5.1 Routing Function

We define a function *Route* which represents the routing algorithm of section 3.1 for an arbitrary number Num_Node of nodes. Num_Node is a natural number, that is a multiple of 4. It computes the path - a list of node numbers - between nodes *from* and *to*.

Definition 17. *Routing Function*

$Route(from,\ to,\ Num\_Node)$ *returns* $Path \stackrel{def}{=}$
  let Rel_addr *be* $(to - from)\ mod\ Num\_Node$ *in:*
  if $Rel\_addr = 0$ **then** $list(from)$
  elsif $0 < Rel\_addr \leq \frac{Num\_Node}{4}$ **then**
    $list(from, Route(Clockwise(from, Num\_Node), to, Num\_Node))$
  elsif $Num\_Node - \frac{Num\_Node}{4} \leq Rel\_addr < Num\_Node$ **then**
   $list(from, Route(Counterclockwise(from, Num\_Node), to, Num\_Node))$
  else
   $list(from,\ Route(Across(from,\ Num\_Node),\ to,\ Num\_Node))$
  endif

where

$Clockwise(from,\ n) \stackrel{def}{=}\ (from + 1)\ mod\ n$

$Counterclockwise(from,\ n) \stackrel{def}{=}\ (from - 1)\ mod\ n$

$Across(from,\ n) \stackrel{def}{=}\ (from + \frac{n}{2})\ mod\ n$

The following properties establish the correctness of function *Route*: a) it terminates; b) it computes a path consistent with the network topology; and c) the number of hops is less than or equal to $\frac{Num\_Node}{4}$.

In ACL2, a new recursive function must be proven to terminate before it can be added to the logic. This is done by exhibiting a measure (an ordinal) that decreases according to a well-formed relation based on the ordinals up to ϵ_0 (chapter 6 of [2]). The measure used for *Route* is $Min[(dest - from)\ mod\ n, (from - dest)\ mod\ n]$.

The second property is divided in three parts. First, we prove that each move is part of the available ones: clockwise, counterclockwise or across. Second, *Route* produces a non-empty path that contains no duplicate. Finally, we prove that a path starts with node *from* and ends with node *to*.

The following theorem states the properties b) and c) on *Route*. Predicates *noDuplicatesp* and *AvailableMovesp* are obvious, and not spelled out for brevity.

Theorem 4. *Correctness of Route*

$\forall\ from,\ to, Num\_Node \in \mathcal{N}$ *such that* $Num\_Node\ mod\ 4 = 0$
  $0 < len(Route(from, to, Num\_Node)) \leq \frac{Num\_node}{4} + 1$
  $\wedge\ noDuplicatesp(Route(from, to, Num\_node))$
  $\wedge\ AvailableMovesp(Route(from, to, Num\_node), Num\_Node)$
  $\wedge\ First(Route(from, to, Num\_Node)) = from$
  $\wedge\ Last(Route(from, to, Num\_Node)) = to$

5.2 Scheduler

In the rest of the paper, we consider that an order is pending at each master (a *no_op* operation standing for the absence of order). Master 0 is given the highest priority, and master $Num\_Node - 1$ the lowest. The pending orders are represented by a list *op_lst* which has the following form:

$$op\_lst = (...\ (i\ op_i\ loc_i\ item_i)\ ...\ (j\ op_j\ loc_j\ item_j)\ ...) \tag{8}$$

where i is a node number, *op* an operation, and *loc* a global address.

The role of the scheduler is to identify all the pending orders that can be concurrently executed, taking into account their priority. The local communications are always executed, removed from *op_lst*, and their results are stored. The other requests involve distant communications, and their route is computed. A priority ordered *travel list* is built, where each travel is a request followed by its route. It has the following form:

$$tl = (... \ (\ (r/w_k \ addr_k \ dat_k)\ k\ n_1\ n_2\ ...f)\ ...) \tag{9}$$

where k is the source node and f is the final node computed by: $f = addr\_k \ div \ ms$. By a simple induction, we prove that Theorem 4 holds for every route in tl.

We define a function *Scheduler* which extracts a set of non-overlapping routes from tl, i.e. such that a node may appear in at most one route. It takes three arguments: 1) the travel list tl; 2) a list *non_ovlp*, initially empty, that contains the non-overlapping communications at the end of the computation; 3) the list *prev*, initially empty, of the nodes used by the communications in *non_ovlp*. Each computation step processes one request route, and adds it to *non_ovlp* if the intersection of its node set with *prev* is empty; then *prev* is updated. For brevity, overlapping communications are dropped in function *Scheduler* below. In the full model, they are stored in another travel list, for later processing.

Definition 18. *Scheduler*
$Scheduler(tl\ non\_ovlp\ prev)\ returns\ Travel\_List \stackrel{def}{=}$
  if $tl = \emptyset$ **then**
    non_ovlp
  else
    if $getroute(car(tl)) \cap prev = \emptyset$ **then**
      $Scheduler(cdr(tl), car(tl) \cup non\_ovlp,$
                $getroute(car(tl)) \cup prev)$
    else
      $Scheduler(cdr(tl), non\_ovlp, prev)$
    endif
  endif

Let *Grab_nodes(tl)* be a function that creates the list of the nodes used by every route in a travel list tl. The correctness of *Scheduler* is expressed by the following theorem:

Theorem 5. *Correctness of Scheduler*
$\forall\ tl,\ noDuplicatesp(grab\_nodes(Scheduler(tl, nil, nil)))$

This theorem is proven by induction, using Theorem 4 on *Route* and its extension to all the routes in tl.

5.3 Traveling Functions

We define a function *Switch* which represents a generic switch component (Figure 4). It takes node numbers as arguments: the four inputs (*from_x*), two commands

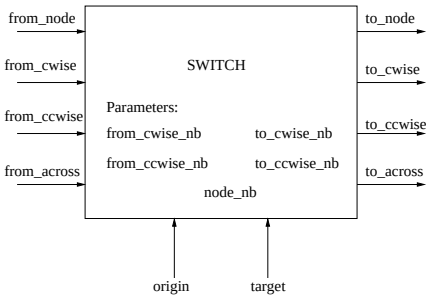

Fig. 4. Generic Switch

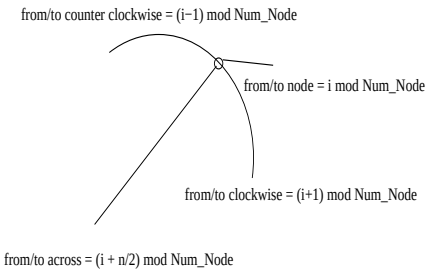

Fig. 5. A step in a travel

(*origin* and *target*) and the parameters. It produces a new value for every output. The switch reads a message on the input selected by the value of *origin*, and writes the message on the output selected by the value of *target*. The other outputs are set to a default "no message" value.

In our model, a message travels on its route r as a result of iterative calls to function *Switch*, until every node of r has been visited. Let i be the current node at a given travel step in route r. *Switch* is called with i as *node_nb*. *origin* and *target* take the previous and next node numbers w.r.t. i in r. The other parameters are numbered as pictured on Figure 5. If i is the first node of r, *origin* is equal to i. If i is the last node of r, *target* is equal to i. The values assigned to the outputs of *Switch*, as a result of executing one travel step, are used in the next call to *Switch* where i is replaced by its successor in r. These calls to *Switch* represent the structure of the interconnected nodes effectively involved in the travel along route r. The set of concurrent travels over the structure are represented by function *Trip*, which takes as arguments a travel list *tl* and the parameter *Num_Node* and executes the travel of every request in the travel list. To validate this function, we check that the theorem below is true if and only if every route in *tl* contains no duplicate and satisfies the predicate *AvailableMovep*. For space reasons, the definitions of functions *Switch* and *Trip* are not given.

Theorem 6. *Correctness of the Interconnection Structure*
$\forall tl, Num\_Node,\ Trip(tl, Num\_Node) = tl$

5.4 Correctness of Distant Communications

Function *Octagon* represents the overall system. It takes as arguments the list *op_lst* containing the orders pending at every node, the two parameters *Num_Node* and *ms* and the global memory *Glob_Mem*. It first recursively calls function *Node* for every order of *op_lst*. Every such call either produces a result, which is stored in a list *LocRes*, or produces a request, which is put, together with its route, in a travel list *tl*. Second, it calls *Scheduler* to extract the non-overlapping communications from *tl*. Then, a first call to *Trip* moves every request to its destination node. Function *Node* is recursively called for each one of the requets to compute

the response of every one of them. The responses are carried back to their respective source node by a second call to *Trip*. Finally, a third recursive call to *Node* is made to compute the result of every response. Function *Octagon* returns the list *LocRes* of the local orders, the list *NetwDone* of the results of the distant orders and the final memory.

Definition 19. *Octagon*
$Octagon(op\_lst, Num\_Node, ms, Glob_Mem)$
$returns(LocRes\ NetwRes\ Memory) \stackrel{def}{=}$
let *(LocRes tl Glob_Mem1)* **be**
Rec-Node-MakeTravelList(op_lst, Nun_Node, ms, Glob_Mem) **in**
tl1 = Scheduler(tl, ∅, ∅)
tl2 = Trip(tl1 N)
let *(RespTl Glob_Mem2)* **be**
Rec-Node-Response(tl2, Glob_Mem1, ms) **in**
RespTl_back = Trip(RespTl, Num_Node)
let *(NetwDone Glob_Mem3)* **be**
Rec-Node-Result(RespTl_back, Glob_Mem2, ms)
list(LocRes NetwDone Glob_Mem3)

To validate this function, we need to prove that Theorem 3 still holds. Let us consider distant communications. The theorem below states the correctness of distant communication in a very general case.

Theorem 7. *Correctness of Octagon for Distant Communications*
$\forall op\_lst, Num\_Node, ms, Glob\_Mem$
$NetwDone(Octagon(op\_lst, Num\_Node, ms, Glob\_Mem))[i] =$
$Memory(Order(op\_lst)[i],$
$get\_local\_mem(Glob\_Mem, Last(Route(op\_lst)[i]), ms))$

Proof. First, Theorem 4 and 5 relieve the conditions of the validity of Theorem 6 which allows us to remove the calls to *Trip*. Then, for every order of *op_lst* three calls to *Node* are made which reconstitute the definition of a transfer, *i.e.* calls to MI_{req}, SI_{ord}, *Slave*, SI_{resp} and MI_{res} □

In practice, we decompose this final proof in a litany of theorems which consider separately the correctness of the returned status, data and memory. We also split read orders from write orders. For instance, we prove that if *op_lst* contains only distant read orders then the memory is not changed.

Theorem 8. *Correctness of the Memory for Read Orders*
$\forall\ op\_lst, Glob\_Mem, ms, Num\_Node$
$all\_distant\_read\_ordersp(op\_lst) \implies$
$GetMemory(Octagon(op\_lst, Num\_Node, ms, Glob\_Mem)) = Glob\_Mem$

Similarly, we prove that every write order is equal to a direct update of the memory.

Theorem 9. *Correctness of the Memory for Write Orders*
$\forall\, op\_lst, Glob\_Mem, ms, Num\_Node, i$
$all\_distant\_write\_ordersp(op\_lst) \wedge 0 \leq i < Num\_Node \implies$
$GetMemory(Octagon(op\_lst, Num\_Node, ms, Glob\_Mem))[i] =$
$Update(Glob\_Mem, Location(Orders(op\_lst)[i]), Items(Orders(op\_lst)[i]))$

6 Conclusion and Future Work

In this paper, we have presented an original approach to model and analyse master/slave based communication architectures in the ACL2 logic. We have illustrated this approach on the *Octagon*, for which we built a functional model that can be executed on test cases. Using the ACL2 theorem prover, the functional correctness of the network routing and scheduling algorithm were established. We proved that tokens travel correctly over this structure: messages eventually reach their expected destination without being modified. In reality, our results hold on a generalization of the *Octagon*: we model an *unbounded* interconnection structure as of Fig. 3, where the number of switches is a multiple of 4.

The model and its proof were developed in three months but the proof can be replayed in less than ten minutes on a Pentium IV at 1.6 GHz with 256 Mb of main memory, under Linux. The overall model [7] contains around one hundred definitions and the proof requires more than two hundred lemmas and theorems.

Most of the human effort was spent on building the foundations for the modeling methodology, ensuring that the functional definitions for the interconnected hardware modules are compositional, and identifying the key theorems. The result is a specification infrastructure for scalable network modules. Thanks to the compositionality of our methodology, most of the functions may be redefined; provided the essential theorems still hold for them, the overall proof is not changed. For instance, the scheduling function may implement a different priority policy: if Theorem 5 still holds on the new scheduling, Theorems 6 and 7 remain valid. Likewise, the routing algorithm of another network structure may be redefined: if it can be proved to satisfy Theorem 4, the final theorems remain valid.

In the near future, we are planning to apply the above ideas to other network on chip designs, in the context of the European "EuroSoC" network.

7 Acknowledgements

We are thankful to J Strother Moore, Warren A. Hunt, Jr, their research group and Matt Kaufmann for their precious help on ACL2 and many discussions. We specially thank Robert Krug for his help on the arithmetic packages.

References

1. W. Roesner: What is Beyond the RTL Horizon for Microprocessor and System Design. Invited Speaker. *Correct Hardware Design and Verification Methods(CHARME)* (2003)

2. M. Kaufmann, P. Manolios and J Strother Moore: Computer-Aided Reasoning: An Approach. Kluwer Academic Publisher (2000)
3. F. Karim, A. Nguyen and S. Dey: An Interconnect Architecture For Networking Systems On Chip. *IEEE Micro* (Sept-Oct 2002) pp. 36–45
4. F. Karim, A. Nguyen, S. Dey and R. Rao : On-Chip Communication Architecture for OC-768 Network Processor. *Design Automation Conference* (2001)
5. J. A. Rowson and A. Sangiovanni-Vincentelli: Interface-Based Design. *Design Automation Conference* (1997)
6. J. Schmaltz: Functional Specification and Validation of the Octagon Network on Chip Using the ACL2 Theorem Prover. *TIMA Technical Report* ISRN TIMA-RR–04/01/02–FR (2004)
7. http://tima.imag.fr/VDS/ProjectValidation.asp (2004)