A Functional Specification and Validation Model for Networks on Chip in the ACL2 Logic[1]

Julien Schmaltz[2] 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 model used to specifiy and validate, in the ACL2 logic, a *system on a chip* communication architecture named *Octagon*. The functional model is briefly introduced before being developed on the case study. We define and validate the routing algorithm, a simple scheduling algorithm and the correctness of read and write operations which includes the proof that messages travel over the network without being modified and eventually reach their expected destination.

1 Introduction

Nowadays, if the design and the verification of *systems on chip* is well supported at the Register Transfer Level (RTL) and below [1], the first design steps are only supported by numeric simulation and *ad hoc* tools. Furthermore, systems increasingly reuse pre-existing modules, which have been intensively verified in isolation, and an essential aspect of the overall functional correctness of systems relies on the correctness of their communications. In this context, our work focuses on the specification of the communications on a chip at a high level of abstraction. It involves generic network components where the number of interconnected modules is *finite*, but not necessary *bounded*.

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*. The ACL2 model is based on the informal descriptions presented in the scientific literature ([4] and [5]). Nervertheless, our model is more general in the sense that it is *parameterized* by the number of nodes and the size of the memory. Our main contribution is the application of theorem proving techniques to on chip communication architectures, which is, to the best of our knowledge, new. The study of communication protocols (regarding the ACL2 community

[1] This paper is adapted from a higher-level paper presented at the FMCAD conference [7] and provides details for an ACL2-literate audience

[2] 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.

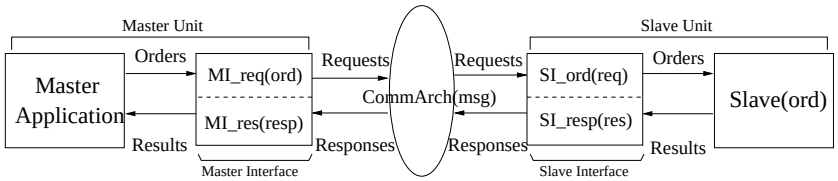

Fig. 1. Formalization of Communications

Moore's work [3] is one example) is somehow related to our work, but not directly. These studies formalize *how data are encapsulated in messages*; we focus here on formalizing *how messages travel* over an interconnect. Our main results are a first model for networks on chip and the development of a specific library of functions and theorems, most of which will be reusable for circuits of similar functionnality.

In the next section, we briefly introduce the functional model 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 model of section two. The functional definition of the *Octagon*, and the main theorems that validate this model are given in section five. We prove the correctness of read and write operations which includes the proof 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 Modeling of Communications

Our model is pictured on Fig. 1. A master starts the communication by sending an *order* to the slave, which replies with a *result*. Communication operations are *orthogonal* to the computation operations [6]. They are separated in two classes of components: the interfaces and the applications. 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.

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*. Master and slave interfaces are not directly connected. A communication medium, bus or network, determines how requests and responses are transfered. The medium is modeled by a function *CommArch* which takes

and returns a *response* or a *request*, *i.e.* a *message*. Communications are modeled by the composition of these functions.

The transfer of an order from the master to the slave application is defined as the composition of MI_{req}, $CommArch$ and SI_{ord}:

Definition 1. *Transmission of an Order via a medium*

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

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

Definition 2. *Transmission of a Result via a medium*

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

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 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 slave interface will modify the address of the original order to satisfy a specific mapping of the slave application addresses. Consequently, the order received by the slave application is not strictly equal to the sent order, but equal *modulo* a given address mapping. This is expressed by some relation $\simeq$ which is defined according to the specific memory structure of a system. If the transmission of an order is correct, then the following is a theorem:

Theorem 1. *Trans_Ord Correctness*

$$\forall\, order,\ trans_ord(order) \simeq order$$

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. If the transmission of a result is correct, then the following is a theorem:

Theorem 2. *Trans_Res Correctness*

$$\forall\, result,\ Trans_res(result) = result$$

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*

$$\forall\, order,\ Transfer(order) \simeq Slave(order)$$

Proof. follows from theorems 1 and 2. □

In the remainder of this paper, we develop this functional style through the definition and the validation, in the ACL2 logic, of the Octagon architecture.

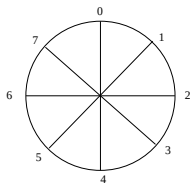

Fig. 2. Basic Octagon Unit

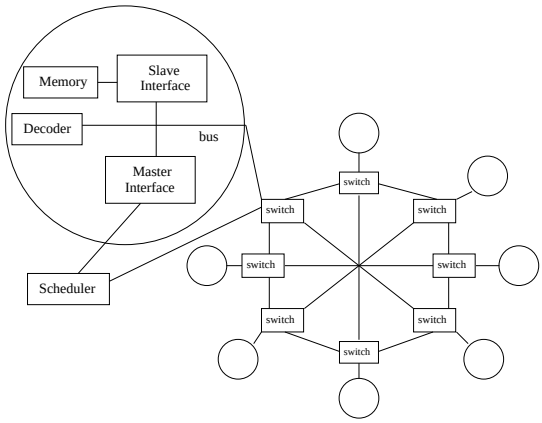

Fig. 3. Node System connected to a switch and the scheduler

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 ([4] and [5]).

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:

$$Rel_addr \;=\; (Packet_addr \;-\; Node_addr)\; mod\; 8 \tag{1}$$

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

- $Rel_addr = 0$, 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 a summary of the definition and the validation of the network in the ACL2 logic. The complete ACL2 model is given as supporting materials to this paper.

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: $local_addr = global_addr \ mod \ ms$.

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

$$i \ = \ global_addr \ div \ ms \tag{2}$$

The address decoder receives a *global* address and determines whether the slave at node *node_nb* should be active or not. It is defined as follows:

Definition 4. *Address Decoder*

```
(defun decoder (global_addr ms node_nb)
  (if (and (< global_addr (* (1+ node_nb) ms))
           ;; less than the first address of next node
           (<= (* node_nb ms) global_addr))
           ;; greater than or equal to the first
           ;; address of the current node.
```

```
      1 ; l/nl=1 for local transfers
    0)) ; l/nl=0 for non local transfers
```

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 5. *Memory*

```
(defun MEMORY (op addr item memo)
  (if (< addr (len memo)) ;; then we are OK and can do op
      (if (equal op 'read)
          (mv-let (dat mem)
                  (mem_read addr memo) ;; = call to nth
                  (mv 'OK dat mem))
        (if (equal op 'write)
            (mv-let (dat mem)
                    (mem_write addr memo item) ;; call to put-nth
                    (mv 'OK dat mem))
          (mv 'INV_OP 'INV_DATA memo)))
    (mv 'INV_ADDR 'INV_DATA memo)))
```

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

$$Glob_Mem = (d_0\ d_1\ ...\ d_{ms-1}\ d_{ms}\ d_{ms+1}\ ...\ d_{num_node \times ms-1}) \tag{3}$$

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 6. *Get Local Memory*

```
(defun get_local_mem (Glob_Mem node_nb ms)
  (firstn ms (nthcdr (* node_nb ms) Glob_Mem)))
```

Example 2. Consider *Glob_Mem* = (a b c d e f), *ms*=2 and Num_Node=3. The memory of node 1 is (firstn 2 (nthcdr (* 1 2) Glob_Mem)) = (firstn 2 (c d e f)) = (c d).

Definition 7. *Update Local Mem*

```
(defun update_local_mem (Glob_Mem memo node_nb ms)
    (append (firstn (* node_nb ms) Glob_Mem)
            (append memo
                    (nthcdr (* (+ 1 node_nb) 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) = (append (a b) (g h) (e f)) = (a b g h e f)

On these functions we prove some useful properties on the length of their result. We also prove that updating the global memory at a given node using the memory list of this node does not change the initial global memory:

```
(defthm update_and_get_local_ok
  (implies (and (equal memo (get_local_mem Glob_Mem node_nb ms))
                (integerp node_nb) (<= 0 node_nb)
                (true-listp Glob_Mem) (NODE_MEM_SIZEp ms)
                (< node_nb (floor (len Glob_Mem) ms)))
           (equal (update_local_mem Glob_Mem memo node_nb ms)
                  Glob_Mem)))
```

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 (*i.e* an operation, a location and a data), 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 communication, the emitted request, the response to the incoming request, and the new value of the global memory.

Definition 8. *Node System*

```
(defun node (op loc dat Glob_Mem ;; pending order and memory
             nw_stat nw_r_dat ;; response from network
             nw_r/w nw_addr nw_dat ;; request from network
             IncomingResponse IncomingRequest ;; validity flags
             node_nb ;; node number
             ms ;; size of local memory
             )
  (if (equal op 'NO_OP) ;; node master is doing nothing
      (if (equal IncomingRequest 1) ;; valid request from ntwk
          (let ((dec (decoder nw_addr ms node_nb)))
```

```
        (mv-let (st dat memo)
                (nw_transfer nw_r/w nw_addr nw_dat Glob_Mem
                             dec node_nb ms)
                (mv 'NO_OP 'NO_DATA ;; no result
                    st dat 'NO_MSG_DATA ;; response to ntwk
                    memo))) ;; new memory
    (if (equal IncomingResponse 1) ;; valid response
        (mv-let (stat r_dat)
                (MI_res nw_stat nw_r_dat) ;; get result
                (mv stat r_dat ;; result to master
                    'NO_MSG_R/W 'NO_MSG_ADDR 'NO_MSG_DATA
                    ;; no response or no request to ntwk
                    Glob_Mem)) ;; the memory is not changed
      (mv 'NO_OP 'NO_DATA 'NO_MSG_R/W
          'NO_MSG_ADDR 'NO_MSG_DATA Glob_Mem)))
;; else the node master is doing a write or read operation
(let ((dec (decoder loc ms node_nb)))
  (if (equal dec 1) ;; local communication
      (mv-let (st dat memo)
              (bus_transfer op loc dat Glob_Mem
                            dec node_nb ms)
              (mv st dat ;; result of the local communication
                  'NO_MSG_R/W 'NO_MSG_ADDR 'NO_MSG_DATA
                  memo))
          ;; else the node sends a request to the ntwk
    (mv-let (r/w addr data)
            (mi_req op loc dat) ;; get request
            (mv 'NO_OP 'NO_DATA ;; no result
                r/w addr data ;; request sent to the ntwk
                Glob_Mem))))))
```

Local communications are represented by function $Bus_transfer$ which is defined similarly to the definitions 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 9. *Network Transfer*

```
(defun nw_transfer (r/w addr data Glob_Mem sl_select node_nb ms)
  (mv-let (op loc dat)
          (si_ord r/w addr data sl_select ms)
          (mv-let (stat dat memo)
                  (memory op loc dat
                          (get_local_mem Glob_Mem node_nb ms))
                  (mv-let (st d)
                          (si_resp stat dat sl_select)
                          (mv st d
                              (update_local_mem Glob_Mem memo
                                                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*. To simplify the reasoning within ACL2, Num_Nodes is defined as $(* \ 4 \ n)$ where n is a positive integer.

Definition 10. *Routing Function*

```
(defun route (from dest n)
  (cond ((or (not (integerp dest))
             (< dest 0)
             (< (- (* 4 n) 1) dest)
             ;; dest must be lower than the number of nodes
             (not (integerp from)) ;; from must be an integer
             (< from 0)
             (< (- (* 4 n) 1) from)
             ;; from must be lower than the number of nodes
             (not (integerp n))
             (<= n 0))
            nil)
        ((equal (- dest from) 0) ;; process at node
         (cons from nil))
        ((and (< 0 (mod (- dest from) (* 4 n)))
              (<= (mod (- dest from) (* 4 n)) n))
         (cons from (route (n_clockwise from n) dest n)))
```

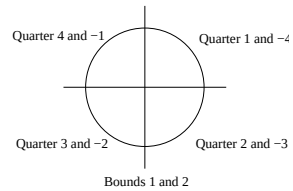

Fig. 4. Decomposition of the Octagon

```
((and (<= (* 3 n) (mod (- dest from) (* 4 n)))
      (< (mod (- dest from) (* 4 n)) (* 4 n)))
 (cons from (route (n_counter_clockwise from n) dest n)))
(t
 (cons from (route (n_across from n) dest n)))))
```

where n_clockwise, n_counter_clockwise and n_across are defined as:

```
(defun n_clockwise (from n)
  (mod (+ from 1) (* 4 n)))

(defun n_counter_clockwise (from n)
  (mod (- from 1) (* 4 n)))

(defun n_across (from n)
  (mod (+ from (* 2 n)) (* 4 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 $\frac{Num_Node}{4}$. 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 measure used to prove that *Route* terminates, is

$$Min[(dest - from)\ mod\ (4 \times n),\ (from\ -\ dest)\ mod\ (4 \times n)]$$

It is generally hard to reason about *mod* in ACL2. In this proof of termination, we use the last arithmetic packages [8] and ten additional lemmas. Once the function is admitted in the logic, we decompose the computation according to the following eight quarters and two bounds (Figure 4):

1. **Quarter 1.** $0 \leq to - from \leq \frac{n}{4}$
2. **Quarter -4.** $-n < to - from \leq -\frac{3n}{4}$
3. **Quarter 2.** $\frac{n}{4} < to - from < \frac{n}{2}$
4. **Quarter -3.** $-\frac{3n}{4} < to - from\ -\frac{n}{2}$

5. **Quarter 3.** $\frac{n}{2} < to - from < \frac{3n}{4}$
6. **Quarter -2.** $-\frac{n}{2} < to - from < -\frac{n}{4}$
7. **Quarter 4.** $\frac{3n}{4} \leq to - from < n$
8. **Quarter -1.** $-\frac{n}{4} \leq to - from < 0$
9. **Bound 1.** $to - from = \frac{n}{2}$
10. **Bound 2.** $to - from = -\frac{n}{2}$

For each case, we prove that *Route* is equivalent to a small function that does not use *mod*. Reasoning is thus simplified.

The following theorem states the correctness of *Route*. Some predicates are obvious, and not spelled out for brevity.

Theorem 4. *Correctness of Route*

```
(defthm CORRECTNESS_OF_ROUTE
   (implies (and (integerp from) (<= 0 from) (< from (* 4 n))
                 (integerp to) (<= 0 to) (< to (* 4 n))
                 (integerp n) (< 0 n))
            (and (consp (route from to n))
                 ;; every node is an integer
                 (all_intp (route from to n))
                 ;; every node number is positive
                 (all_pos_intp (route from to n))
                 ;; every route contains no duplicate
                 (no-duplicatesp (route from to n))
                 ;; every node is less than the maximum of nodes
                 (all_inf_np (route from to n) (* 4 n))
                 ;; a route is made of available moves
                 (AvailableMovep (route from to n) n)
                 ;; the first node is the starting node
                 (equal (car (route from to n)) from)
                 ;; the last node is the final node
                 (equal (car (last (route from to n))) 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:

$$op_lst = (... \ (i \ op_i \ loc_i \ item_i) \ ... \ (j \ op_j \ loc_j \ item_j) \ ...) \tag{4}$$

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:

$$tl = (... \ (\ (r/w_k \ addr_k \ dat_k) \ k \ n_1 \ n_2 \ ...f) \ ...) \tag{5}$$

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 11. *Scheduler*

```
(defun scheduler (tl non_ovlp_r prev)
 ;; extracts non overlapping communications of tl
 (if (endp tl)
    (rev non_ovlp_r)
  (let ((route_i (cdr (car tl))))
      (if (no_intersectp route_i prev)
          (scheduler (cdr tl)
                     (cons (car tl) non_ovlp_r)
                     (append route_i prev))
         (scheduler (cdr tl) non_ovlp_r prev)))))
```

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*

```
(defthm all_no_duplicatesp_scheduler
  (implies (all_no_duplicatesp tl)
           (no-duplicatesp (grab_nodes (scheduler tl nil prev)))))
```

Proof. The proof requires three lemmas. First, we prove that the scheduler produces a travel list in which every route is unique (but two routes may have nodes in common):

```
(defthm all_no_intersectp_scheduler_non_tail
  (all_no_intersectp_routep (scheduler tl nil prev)))
```

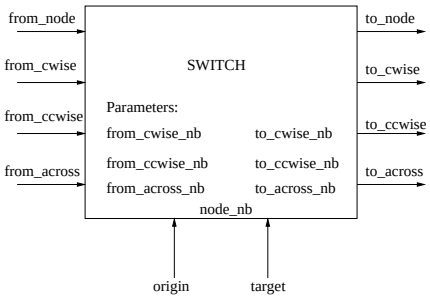

Fig. 5. Generic Switch

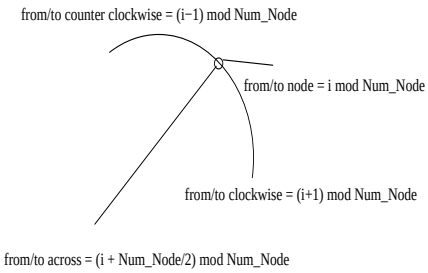

Fig. 6. A step in a travel

Then, we prove that if every route of a travel list *tl* contains no duplicate, then every route of the travel list produced by the function *scheduler* contains also no duplicate:

```
(defthm no_dupli_tl_=>_no_dupli_scheduler
  (implies (all_no_duplicatesp tl)
           (all_no_duplicatesp (scheduler tl nil prev))))
```

Finally, we prove that if every route is unique and if every route contains no duplicate in a travel list *tl*, then *(Grab_nodes tl)* returns a list without duplicate:

```
(defthm all_no_dupli_and_all_no_inter_route_=>_no_dupli_grab_nodes
  (implies (and (all_no_intersectp_routep l)
                (all_no_duplicatesp l))
           (no-duplicatesp (grab_nodes l))))
```

□

5.3 Traveling Functions

We define a function *Switch* which represents a generic switch component (Figure 5). It takes as arguments: the four inputs (*from_x*), two commands (*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 NIL.

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 6. 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. (For space reasons, the definitions of functions *Switch* and *Trip* are not given.) To validate this function, we first prove that if every route in *tl* contains no duplicate and satisfies the predicate *AvailableMovep* then *Trip* does not modify the message:

Theorem 6. *Correctness of the Interconnection Structure 1*

```
(defthm correctness_of_Trip
  (implies (and (all_no_duplicatesp tl)
                (all_pos_intp_route_lstp tl) (all_int_routep tl)
                (all_inf_routep tl (* 4 N))
                (all_availableMovep_routep tl N)
                (all_true-listp tl) (integerp N) (< 0 N)
                ;; tl is a travel list
                (tlp tl))
           (equal (trip tl N) tl)))
```

Proof. By a simple induction on *tl*. A suitable induction scheme is automatically found by ACL2. Some ACL2 heuristics prevent the opening of some recursive predicates and thus an expand hint is required. 8 simple rewrite rules are also needed in addition to ACL2 pre-existing rules. □

Then, we check that messages are lost if every route in *tl* is not valid:

Theorem 7. *Correctness of the Interconnection Structure 2*

```
(defthm correctness_of_Trip_not
  (implies (and (all_no_duplicatesp tl)
                (all_pos_intp_route_lstp tl) (all_int_routep tl)
                (all_inf_routep tl (* 4 N))
                ;; routes are not valid
                (all_not_available_routep tl N)
                (integerp N) (< 0 N) (all_true-listp tl) (tlp tl))
           (all_nil_msg (trip tl N))))
```

Proof. This proof is a little more complex. The proof requires 2 inductions, generalization, destructor elimination and 3 additional rewrite rules including Theorem 6. We also use an expand hint similar to Theorem 6. □

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 (function *ComputeResponses*). The responses are carried back to their respective source node by a second call to *Trip*. Finally, a third recursive call to *Node* computes the result of every response (function *ComputeRes*). 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 12. *Octagon*

```
(defun Octagon (op_lst N ms Glob_Mem)
  ;; model of the complete network: nodes connected to Octagon
  ;; runs the Ntwk once, returns loc_done nw_done and memory
  (mv-let (loc_done nw_op Glob_Mem1)
          ;; collect messages and execute the local operations
          ;; we also get the non local requests
          (collect_msg op_lst nil nil Glob_MEM ms)
          ;; then we compute the travel list
          (let* ((tl (make_travel_list nw_op nil ms N))
                 ;; we extract the set of non-overlapping comms
                 (novlp (scheduler tl nil nil))
                 ;; we move every request to their destination
                 (tl_at_dest (trip novlp N)))
            (mv-let (cr_lst Glob_Mem2)
                    ;; we compute the response of every request
                    (ComputeResponses tl_at_dest Glob_Mem1
                                      ms nil)
                    ;; move responses back to their source node
                    (let ((tl_back (trip cr_lst N)))
                      (mv-let (nw_done Glob_Mem3)
                              (ComputeRes tl_back Glob_Mem2
                                          ms nil)
                              (mv loc_done nw_done
                                  Glob_Mem3)))))))
```

To validate this function, we need to prove a theorem equivalent to Theorem 3,*i.e.* to prove that read or write operations exhibit the same behaviour through the Octagon as they would have through direct interaction with the memory. We decompose this final proof into 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*

```
(defthm mem_ok_read_Octagon
  (implies (and (all_read_op_lstp op_lst)
                (all_non_loc_op_lstp op_lst ms)
                (all_node_nb_validp op_lst (* 4 N))
                (all_address_validp op_lst (* 4 N) ms)
                (equal (len Glob_Mem) (* (* 4 N) ms))
                (integerp N) (< 0 N) (true-listp Glob_Mem)
                (NODE_MEM_SIZEp ms))
           (equal ;; final memory
                  (mv-nth 2
                          (Octagon op_lst N 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*

```
(defthm mem_ok_write_octagon
  (implies (and (all_write_op_lstp op_lst)
                (all_non_loc_op_lstp op_lst ms)
                (all_node_nb_validp op_lst (* 4 N))
                (all_address_validp op_lst (* 4 N) ms)
                (equal (len Glob_Mem) (* (* 4 N) ms))
                (integerp N) (< 0 N) (true-listp Glob_Mem)
                (NODE_MEM_SIZEp ms))
           (equal ;; final memory
                  (mv-nth 2 (Octagon op_lst N MS Glob_Mem))
                  ;; correct modification of the memory
                  (good_mem_write
                   (scheduler
                    (make_travel_list
                     (mv-nth 1 (collect_msg op_lst nil nil
                                            Glob_Mem ms))
                    NIL ms N)
                   nil nil)
                  Glob_Mem ms))))
```

where *good_mem_write* is the following function:

```
(defun good_mem_write (req_lst mem ms)
  ;; in case of good write requests, the location
  ;; is changed through a call to put-nth
  (if (endp req_lst)
```

```
 mem
(good_mem_write
 (cdr req_lst)
 (put-nth (global_addr (nth 1 (caar req_lst))
                       (last_route req_lst)
                       ms)
          (nth 2 (caar req_lst))
          mem)
 ms)))
```

6 Conclusion and Future Work

In this paper, we have presented a functional model for on chip communications. We have illustrated this approach on the *Octagon*. The functional correctness of the network routing and scheduling algorithm were established. We proved the correctness of read and write operations which includes the proof 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 contains around one hundred definitions and the proof requires more than two hundred lemmas and theorems.

Thanks to our decomposition of the communications, most of the functions may be redefined to suit the characteristics of other design decisions. 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.

The work reported here is only a first step. Extensions are required to take into account the full complexity of on chip communications. For instance, the scheduling algorithm of this paper considers only a circuit switched mode, and packet switching algorithms will have to be considered. We are also trying to extend our model (*i.e.* Fig. 1) so that it can take protocols into account. For instance, we are working on the formalization of Ethernet in the *spirit* of this paper.

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 also 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. J Strother Moore: A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol, *Formal Aspects of Computing* (1993)
4. F. Karim, A. Nguyen and S. Dey: An Interconnect Architecture For Networking Systems On Chip. *IEEE Micro* (Sept-Oct 2002) pp. 36–45
5. F. Karim, A. Nguyen, S. Dey and R. Rao : On-Chip Communication Architecture for OC-768 Network Processor. *Design Automation Conference* (2001)
6. J. A. Rowson and A. Sangiovanni-Vincentelli: Interface-Based Design. *Design Automation Conference* (1997)
7. J. Schmaltz and D. Borrione: A Functional Approach to the Formal Specification of Networks on Chip, in *Proc. of the 5th Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, (2004)
8. R. Krug, W. A. Hunt and J Moore: Linear and Nonlinear Arithmetic in ACL2. *Correct Hardware Design and Verification Methods(CHARME)* (2003)