# Towards A Formal Theory of On Chip Communications in the ACL2 Logic

Julien Schmaltz
Saarland University
Institute for Computer Architecture
FR 6.2 Informatik, Postfach 151150
D-66041 Saarbrücken, Germany
julien@cs.uni-sb.de

Dominique Borrione
Joseph Fourier University
TIMA Laboratory - VDS Group
46, avenue Felix Viallet,
38031 Grenoble, France
Dominique.Borrione@imag.fr

## ABSTRACT

This paper is devoted to the expression of a formal theory of communication networks in the ACL2 logic. More precisely, we have developed a generic model called *GeNoC*, in a general mathematical notation, with the use of quantification over variables as well as over functions. We present here its expression in the first order quantifier free logic of the ACL2 theorem prover. We describe our systematic approach to cast it into ACL2, especially how we use the encapsulation principle to obtain a systematic methodology to specify and validate on chip communications architectures. We summarize the different instances of *GeNoC* developed so far in ACL2, some come from industrial designs. We illustrate our approach on an XY routing algorithm.

## Categories and Subject Descriptors

F.0 [**Theory of Computation**]: General; B.7.2 [**Integrated Circuits**]: Design Aids

## General Terms

Communication theory, design and verification

## Keywords

network on a chip, formal theory, theorem proving

## 1. INTRODUCTION

The design of complex systems on a chip (SoC) relies on the integration of pre-existing modules. In this framework, the overall behavior of SoC's highly depends on the interconnect structure. Its design and the verification of the communication architecture become crucial [12].

The principal verification efforts regarding embedded communication architectures are the following. Concerning protocols dedicated to bus architectures, Roychoudhury *et al.* use the SMV model checker to debug an academic implementation of the AMBA AHB protocol [7]. Their model is written at the register transfer level and without any parameter. Roychoudhury *et al.* detect a livelock scenario that comes more from their own arbiter than the protocol itself. More recently, Amjad [1] used a model checker, implemented in the HOL theorem prover, to verify the AMBA protocols APB and AHB and their composition in a single system. Using model checking, safety properties are verified on each protocol. The HOL tool is used to verify their composition. The model is also at a low level of abstraction and without any parameter. Regarding networks on a chip (NoC) little work has been done about their formal verification. Gebremichael *et al.* [3] have recently specified the Æthereal protocol of Philips in the PVS logic. The main property they verified is the absence of deadlock for an arbitrary number of masters and slaves.

On the one hand, these studies consider design at the register transfer level (RTL). The current trend in the SoC design community is to raise the level of abstraction [12]. On the other hand, these studies are dedicated to particular applications. To verify another communication network, one has to formalize and prove everything *again*. Indeed, there is no *formal theory* of communication networks. Most textbook (*e.g.* [2]) *describe* architectures in an informal manner.

The objective of our research is to formalize the different concepts that belong to communication architectures, *i.e.* to define a formal theory for communication networks. We express this theory in a classical mathematical notation. Then, one can cast it into her/his favorite tool.

A first step towards this theory has been achieved in Schmaltz's Ph.D. thesis [8]. The main contribution of this work is the definition of a *generic network on a chip* (*GeNoC*) model. It is defined as the composition of key components (routing, scheduling and interfaces). We have identified the essential properties inherent in each one of them. The proof of the overall correctness of *GeNoC* is directly deduced from these constraints. Hence, this correctness is preserved for any *particular* network architecture, provided its components satisfy the constraints.

We briefly present the general theory in section 2. This paper focusses on how we embed this theory in the ACL2 logic. For instance, the mathematical notations involve quantification over functions which is elegantly expressed using the encapsulation principle and the derived inference rule "functional instantiation". Section 3 presents the strategy we used to express *GeNoC* in the ACL2 logic. Sections 4 to 7


*ACL2* '06 Seattle, Washington USA

expose the ACL2 definition of the components of *GeNoC*. We show concrete instances of *GeNoC* in section 8. Section 9 concludes the paper.

# 2. A GENERIC NETWORK ON CHIP

To treat the different communication architectures in a single formalism, we generalize them to a unique model explained in the next subsection. After that, we describe rapidly function *GeNoC* and give the general expression of its correctness.

## 2.1 A Unique Communication Model

Consider the general communication model of Figure 1. An arbitrary, but finite, number of *nodes* are connected to some communication architecture. The latter represents the interconnection structure, *e.g.* bus or network. It comprises topologies, routing algorithms and scheduling policies. Our model makes no assumption on these components. As proposed by Rowson and Sangiovanni-Vincentelli [6], each node is separated into an *application* and an *interface*. The latter is connected to the communication architecture. Interfaces allow applications to communicate using protocols. Any interface-application pair matches the layers of the OSI model. Interfaces generally refer to layers 1 to 4; applications to layers 4 to 7. Layer 4 is a boundary and can be part of either interfaces or applications. To distinguish between interface-application and interface-interface communications, an interface and an application communicate using *messages*; two interfaces communication using *frames*.

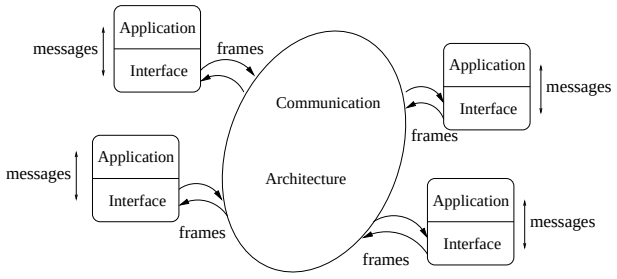


**Figure 1: Communication Model**

Applications represent the computational and functional aspects of nodes. They are either active or passive. Typically, active applications are processors and passive applications memories. We consider that each node contains one passive and one active application, *i.e.* each node is capable of sending and receiving frames. As we want a general model, applications are not considered *explicitly*: passive applications are not actually modeled, and active applications are reduced to the list of their pending communication operations. We focus on communications between distant nodes. We suppose that in every communication, the destination node is different from the source node.

## 2.2 Overview of *GeNoC*

Function *GeNoC* represents a generic communication architecture. This architecture has an arbitrary topology, routing algorithm and switching technique. Function *GeNoC* represents the transfer of messages from their source to their destination. Its main argument is the list of messages emitted at source nodes. It returns the list of the results received at destination nodes. Its definition mainly relies on the following functions:

1. Interfaces are represented by two functions; one function, *send*, to inject frames on the network, and one function, *recv*, to receive frames,
2. the routing algorithm and the topology are represented by function *Routing*,
3. the switching technique is represented by function *Scheduling*.

These functions are generic in the sense that they do not have an explicit definition. They are only defined by a number of properties, called *proof obligations* or simply *constraints*.

**Interfaces.** Function *send* represents the encapsulation of a message into a frame. Function *recv* represents the decoding of this frame to recover the emitted message. The main constraint associated to these functions expresses that a receiver should be able to extract the encoded information, *i.e.* the composition of function *recv* with function *send* ($recv \circ send$) is the identity function.

**Routing Algorithm.** The routing algorithm is represented by the successive application of unitary moves. For each pair made of a source and a destination, the routing function computes *all* possible routes allowed by the unitary moves. The main constraint associated to the routing function expresses that each route from a source $s$ to a destination $d$ effectively starts in $s$ and uses only existing nodes to end in $d$.

**Switching Technique.** The scheduling policy participates in the management of conflicts that appear on the network. It defines the set of communications that can be performed *at the same time*. Formally, these commutations satisfy an *invariant*. Scheduling a communication, *i.e.* adding it to the current set of authorized communications, must preserve the invariant, for all times and in any admissible state of the network. The invariant is specific to the scheduling policy. In our formalization of the scheduling policy, the existence of this invariant is assumed but not explicitly represented. From a list of requested communications, the scheduling function extracts a sub-list that satisfies the invariant. The rest make up the list of delayed communications.

**Function** *GeNoC*.' Function *GeNoC* is pictured in Fig. 2. It takes as arguments the list of requested communications and the characteristics of the network. It produces two lists as results: the messages received by the destination of successful communications and the aborted communications. In the remainder of this section, we detail the basic components of the model.

The main input of *GeNoC* is a list $\mathcal{T}$ of *transactions* of the form $t = (id\ A\ msg_t\ B)$. Transaction $t$ represents the intention of application $A$ to send a message $msg_t$ to application $B$. $A$ is the *origin* and $B$ the *destination*. Both $A$ and $B$ are members of the set of nodes, *NodeSet*. Each transaction is uniquely identified by a natural *id*. Valid transactions are recognized by predicate $\mathcal{T}_{lstp}(\mathcal{T}, NodeSet)$.

Briefly, function *GeNoC* works as follows. For every message in the initial list of transactions, it computes the corresponding frame using *send*. Each frame together with its *id*, *origin* and *destination* constitutes a *missive*. A missive is valid if the ids are naturals (with no duplicate); the origin

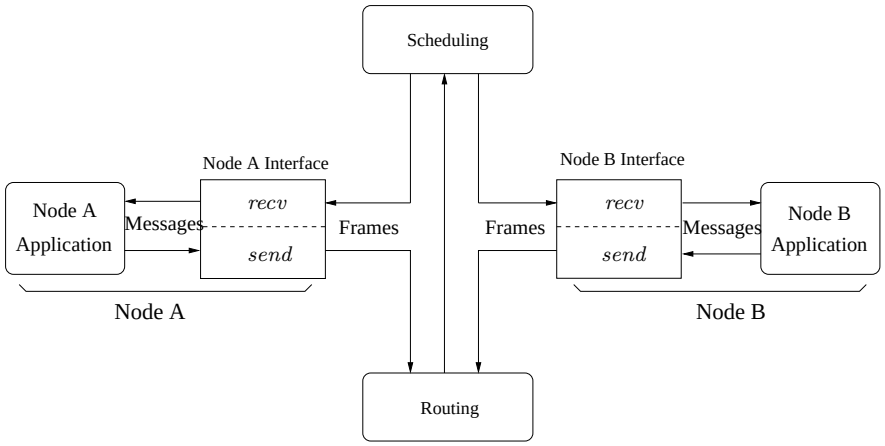


**Figure 2: *GeNoC*: A generic network on chip model**

and the destination are members of *NodeSet*. A valid list, $\mathcal{M}$ of missives is recognized by predicate $\mathcal{M}_{lstp}(\mathcal{M}, NodeSet)$. Then, *GeNoC* computes the routes of the missives and schedules them using functions *Routing* and *Scheduling*. To keep our model general, function *Routing* computes a list of routes for every missive. If the routing algorithm is deterministic, this list has only one element. Once routes are computed, a *travel* denotes the list composed of a frame, its *id* and its list of routes. A list $\mathcal{V}$ of travels is valid if the ids are naturals (with no duplicate). Such a list is recognized by predicate $\mathcal{V}_{lstp}(\mathcal{V})$. The results of the scheduled travels are computed by calling *recv*. The *delayed* travels are converted back to missives and constitute the argument of a recursive call to *GeNoC*. To make sure that this function terminates, we associate to every node a *finite* number of attempts. At every recursive call of *GeNoC*, every node with a pending transaction will consume one attempt. The *association list* *att* stores the attempts and $att[i]$ denotes the number of remaining attempts of the node $i$. Function $SumOfAtt(att)$ computes the sum of the remaining attempts of the nodes and is used as the decreasing measure of parameter *att*. Function *GeNoC* halts if every attempt has been consumed. The first output list $\mathcal{R}$ contains the results of the completed transactions. Every result $r$ is of the form *(id B* $msg_r$*)* and represents the reception of a message $msg_r$ by its final destination $B$. Transactions may not run to completion (*e.g.* due to network contention). The second output list of *GeNoC* is named *Aborted* and contains the cancelled transactions.

Function *GeNoC* is considered correct if every non aborted transaction $t =$ *(id A msg B)* is completed in such a way that $B$ effectively receives *msg*. Formally, we prove that for every final result $r$, there is a unique initial transaction $t$ such that $t$ has the same *id* and *msg* as $r$.

$$\forall rst \in \mathcal{R}, \exists! t \in \mathcal{T}, \left\{ \begin{array}{ll} & Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge & Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge & Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{array} \right. \quad (1)$$

This formula is proved a theorem using the proof obligations associated to each component. These proof obligations have often the same structure. For all elements produced by some function (here function *GeNoC*) we look for a unique element in the principal argument of that function (here the transactions) such that both elements satisfy a given property. We do not go further into the mathematical translation. In the next section, we explain how we translate it to the ACL2 logic. Then, we give all ACL2 definitions and constraints about *GeNoC*, as well as its proof of correctness. The mathematical model has been published elsewhere [8].

# 3. MODELING PRINCIPLES

Functions *Routing*, *Scheduling*, *recv* and *send* are not defined but *constrained* to satisfy a list of properties. In the following subsection, we show how to use the encapsulation principle to express this second order quantification. Using functional instantiation, ACL2 can generate the proof obligations that must be discharged by a particular instance of a component. We show how to systematically use that rule for design verification.

## 3.1 Encapsulation of the Constraints

Function *send* takes a message as a unique argument and returns a frame. No assumption is made on the definition domains, $\mathcal{D}_{msg}$ and $\mathcal{D}_{frm}$ of messages and frames. Functions *send* and *recv*, in the ACL2 logic, are functions taking one argument and returning one argument. They have the following signatures:

$$((send\ *) \Rightarrow *)$$
$$((recv\ *) \Rightarrow *)$$

The main constraint on these functions is that their composition is an identity. This is expressed by the following proof obligation:

PROOF OBLIGATION 1. **Interface Correctness**

```
(defthm InterfaceCorrectness
    ;; recv ∘ send(msg) = msg
   (equal (recv (send msg)) msg))
```

For technical reasons, two other constraints are associated to function *send*. The first one states that if nothing has to be sent, function *send* returns the empty list (constraint *send*-nil). The second constraint states that if the message to be sent is not the empty list, function *send* does not return the empty list (constraint *send*-not-nil).

The complete encapsulate event regarding the interfaces is as follows:

```
(encapsulate
  (((send *) ⇒ *)
   ((recv *) ⇒ *))
  (local (defun send (msg) msg)) ;; local witness
  (local (defun recv (frm) frm)) ;; local witness
  (defthm InterfaceCorrectness
   (equal (recv (send msg)) msg))
  (defthm send-nil
   (not (send nil)))
  (defthm send-not-nil
   (implies msg (send msg))))
```

Using the functional instantiation inference rule, ACL2 generates - and tries to prove - the proof obligations associated to particular definitions of *recv* and *send*. Let us redefine these functions outside the encapsulate. Consider function $send_e$ that starts a communication by sending a constant bit list to synchronize with a receiver. Let this constant be `*start*` = `(0 1 0 1 0 1 0 1)`. To satisfy constraint *send*-`nil`, this function returns `nil` if its input message is `nil`. Its definition is the following:

```
(defun send_e (msg)
   (if (not msg)
      nil
    (append *start* msg))
```

Function $recv_e$ reads a bit list `lst`. If this list is empty, it returns `nil`. If the first 8 bits equal `*start*`, it returns `lst` less these first 8 bits. Otherwise, it consumes one bit and looks for `*start*` in the rest of `lst`. Its definition is:

```
(defun recv_e (lst)
   (if (endp lst)
      nil
     (if (equal (firstn 8 lst) *start*)
        (nthcdr 8 lst) ;; lst less *start*
      (recv_e (cdr lst)))))
```

The proof obligations associated to these two definitions can be automatically generated (and proved) by ACL2. The principle is to prove some property (the constant `t` for instance) and to give a hint to ACL2 that forces it to use the properties of the encapsulate above. We ask ACL2 to prove the following theorem:

```
(defthm check-instance-interface
 t                    ; we prove "true"
 :rule-classes nil    ; no rule is generated
 :hints (("GOAL"
     ; we force ACL2 to use InterfaceCorrectness
     ; by substituting recv by recv_e
     ; and send by send_e
     :use
      (:functional-instance InterfaceCorrectness
                            (recv recv_e)
                            (send send_e)))))
```

A similar approach is taken to check if concrete designs of functions *Routing* or *Scheduling* are valid instances of their generic counterparts. The encapsulate event about these remaining components are described in the next section.

## 3.2 Removing Quantifiers

The ACL2 logic is generally considered quantifier free. The formulae presented in the previous section do not translate directly into ACL2. The principle is to express quantifiers by recursive functions. Let us consider the formula $\forall x \in E, p(x)$, which means that all elements in set $E$ satisfy predicate $p$. In ACL2, we rather consider a list, the elements of which are in $E$. We define a function $f_p$ which verifies that all elements of a list satisfy $p$. The definition of $f_p$ is the following[1]:

$$f_p(l) \triangleq \begin{cases} t & \textbf{if } l = \epsilon \\ p(e) \wedge f_p(l') & \textbf{otherwise } l = e.l' \end{cases} \quad (2)$$

Let $l \subseteq_l E$ mean $l$ is a list, the elements of which are in set $E$. Property $\forall x \in E, p(x)$ becomes $\forall l, l \subseteq_l E, f_p(l)$. In the ACL2 syntax, this is expressed by an implication:

```
(defthm foo
   (implies (Ep l) (f_p l)))
```

where `Ep` is a predicate that recognizes a list, the elements of which are members of $E$.

More generally, the main formulae of *GeNoC* express properties about a list $L$, and the result $\mathcal{F}(L)$ of the application of a function to that list. These properties express that for all elements $e'$ of a list $\mathcal{F}(L)$ , there exists a unique element $e$ of $L$ such that $e$ and $e'$ satisfy some property $\psi(e, e')$. The formula takes the following form:

$$\forall e' \in_l \mathcal{F}(L), \exists! e \in_l L, \psi(e, e') \quad (3)$$

Lists $L$ and $\mathcal{F}(L)$ are lists of missives, travels, transactions, *etc.* Formula $\psi$ always involves the equality between the identifiers of $e$ and $e'$. The uniqueness of element $e$ is ensured by the type information that guarantees the uniqueness of the identifiers of elements of $L$ (resp. $\mathcal{F}(L)$). Filtering list $L$ according to the identifiers of $\mathcal{F}(L)$, one obtains a list that can be compared with $\mathcal{F}(L)$ element by element.

The filtering operator is illustrated as follows. If $\mathcal{V}$ is a list of travels, $\mathcal{V}/ids$ denotes a sublist of $\mathcal{V}$, which is the result of filtering $\mathcal{V}$ according to some identifiers *ids*.

EXAMPLE 1. *If* $\mathcal{V}$ *is*

$$\begin{array}{ll} ( & (123\ m_1\ (1\ 3\ 9)) \\ & (212\ m_2\ (12\ 4\ 25)) \\ & (313\ m_3\ (1\ 12\ 3)) \quad ) \end{array}$$

*then* $\mathcal{V}/(123\ 313)$ *is*

$$\begin{array}{ll} ( & (123\ m_1\ (1\ 3\ 9)) \\ & (313\ m_3\ (1\ 12\ 3)) \quad ) \end{array}$$

Let $f_\psi$ be a Boolean function over two argument lists. $f_\psi$ returns $t$ if the arguments have equal length and property $\psi$ holds pairwise on their corresponding elements; otherwise, $f_\psi$ returns *nil*. The definition of $f_\psi$ is:

$$f_\psi(l_1, l_2) \triangleq \begin{cases} t & \textbf{if } l_1 = \epsilon \wedge l_2 = \epsilon \\ nil & \textbf{if } l_1 \neq \epsilon \wedge l_2 = \epsilon \\ & \quad \vee\, l_1 = \epsilon \wedge l_2 \neq \epsilon \\ \psi(e, e') \wedge f_\psi(l_1', l_2') & \textbf{otherwise } l_1 = e.l_1' \\ & \quad \wedge\, l_2 = e'.l_2' \end{cases}$$

[1]For the existential quantifier, the conjunction is replaced by a disjunction.

Let $\mathcal{D}_L$ be the definition domain of list $L$. Expressions of the form 3, "for all $e'$ of $\mathcal{F}(L)$, there exists a unique $e$ of $L$ such that $\psi(e, e')$", translate to "for all lists $L$ of $\mathcal{D}_L$, function $f_\psi$ applied to list $\mathcal{F}(L)$ and to $L$ filtered by the identifiers of $\mathcal{F}(L)$ is always true". That is expressed as:

$$\forall L \subseteq_l \mathcal{D}_L, f_\psi(\mathcal{F}(L), L/\mathcal{F}(L)_{\lfloor id}) \quad (4)$$

Finally, the universal quantifier is replaced by an implication and we get the following form:

$$L \subseteq_l \mathcal{D}_L \Rightarrow f_\psi(\mathcal{F}(L), L/\mathcal{F}(L)_{\lfloor id}) \quad (5)$$

In the ACL2 syntax, the left hand side of the implication is translated to the characteristic function of domain $\mathcal{D}_L$, noted $\mathcal{D}_L$-p. Let `filter` be the ACL2 function implementing the filtering operator, and `ids` be the function collecting the identifiers, one obtains the following ACL2 code:

```
(defthm bar
   (implies (D_L-p L)
            (f_ψ  (F L)
                (filter L (ids (F L))))))
```

# 4. NODES AND PARAMETERS

We now describe all functions and theorems that form the encapsulation event for the definition of the nodes and the parameters.

Nodes are defined on an arbitrary domain, *GenNodeSet*. A list of elements of that domain is recognized by predicate *NodeSetp*, which is a constrained function. The set of nodes of a particular network is noted *NodeSet*. It is generated from parameters *pms* defined on an arbitrary domain *GenParams* and function *NodeSetGen*. Valid parameters are recognized by predicate *ValidParamsp* and constitute the generating base for *NodeSet*. The functionality of *NodeSetGen* is as follows:

$$NodeSetGen : GenParams \rightarrow \mathcal{P}(GenNodeSet) \quad (6)$$

These functions are valid if, for all parameters recognized by predicate *ValidParamsp*, every element produced by function *NodeSetGen* belongs to domain *GenNodeSet* (*i.e.* satisfies predicate *NodeSetp*):

PROOF OBLIGATION 2. **Definition of** *NodeSet*.

```
(defthm nodeset-generates-valid-nodes
  (implies (ValidParamsp pms)
           (NodeSetp (NodesetGenerator pms))))
```

Finally, we need to prove that, for each particular instance of predicate *NodeSetp*, any sublist of a valid list of nodes is also a valid list of nodes

PROOF OBLIGATION 3. **Sublists of Valid Node Lists.**

```
(defthm subsets-are-valid
  (implies (and (NodeSetp x) (subsetp y x))
           (NodeSetp y)))
```

# 5. ROUTING ALGORITHM

We now describe the function definitions and theorems for the routing module of *GeNoC*. The correctness of routes is not particular to a network. In the next subsection, we define the general predicates that will be used for any routing algorithm. Then, we give the constraints associated with the routing function.

## 5.1 Route Validity

A route $r$ is correct according to some missive $m$ if (1) the first element of $r$ equals the origin of $m$; (2) the last element of $r$ equals the destination of $m$; (3) each node of $r$ is a member of the set *NodeSet* of the existing nodes. The lengh of any route must be greater than 2. Among these properties, one only depends on *NodeSet*. To avoid free variables, we state it in a separate predicate. The other properties are defined as follows:

```
(defun ValidRoutep (r m)
  (and (equal (car r) (OrgM m))
       (equal (car (last r)) (DestM m))
       (<= 2 (len r))))
```

Function `CheckRoutes` takes a list of routes, a missive and the set `NodeSet`. It checks that any route of the list of routes satisfies `ValidRoutep` and is a member of `NodeSet`.

```
(defun CheckRoutes (routes m NodeSet)
  (if (endp routes)
      t
    (let ((r (car routes)))
      (and (ValidRoutep r m)
           (subsetp r NodeSet)
           (CheckRoutes (cdr routes) m NodeSet)))))
```

Predicate `CorrectRoutesp` checks travels correctness according to missives, *i.e.* routes associated to some travel $v$ satisfies predicate `CheckRoutes` for some missive $m$ such that $v$ and $m$ have the same identifier and the same frame. We also check that the list of travels and the list of missives have the same length.

```
(defun CorrectRoutesp (V M NodeSet)
  (if (endp V)
      (if (endp M)
          t ;; len(M) = len(V)
        nil)
    (let* ((tr (car V))
           (msv (car M))
           (routes (RoutesV tr)))
      (and (CheckRoutes routes msv NodeSet)
           (equal (IdV tr) (IdM msv))
           (equal (FrmV tr) (FrmM msv))
           (CorrectRoutesp (cdr V)
                           (cdr M) NodeSet)))))
```

This predicate implies that converting the travel list $\mathcal{V}$ to a missive list produces $\mathcal{M}$.

```
(defthm correctroutesp-=>-tomissives
     (implies (and (CorrectRoutesp V M NodeSet)
                   (Missivesp M NodeSet)
                   (Vlstp V))
              (equal (ToMissives V) M)))
```

## 5.2 Generic Routing Function

The generic routing function takes two arguments: a missive list and the existing nodes. It returns a travel list. Its signature is the following:

```
(((Routing * *) => *))
```

The local witness of the `encapsulate` simply corresponds to routing in a bus. There is only one route made of the

origin and the destination. In the following definition, functions `IdM`, `FrmM`, `OrgM`, `DestM` are the accessors of the various components of a missive: identifier, frame, origin, destination.

```
;; local witness
(local (defun route (M)
         (if (endp M)
             nil
           (let* ((msv (car M))
                  (Id (IdM msv))
                  (frm (FrmM msv))
                  (org (OrgM msv))
                  (dest (DestM msv)))
             (cons (list Id frm
                         (list (list org dest)))
                   (route (cdr M)))))))
(local (defun routing (M NodeSet)
         (declare (ignore NodeSet))
         (route M)))
```

The main constraint on function `Routing` states that it must satisfy predicate `CorrectRoutesp`.

Proof Obligation 4. **Routing Correctness**

```
(defthm Routing-CorrectRoutesp
  (let ((NodeSet (NodeSetGenerator pms)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp pms))
             (CorrectRoutesp (Routing M NodeSet)
                             M NodeSet))))
```

Another constraint checks that this function outputs a valid travel list.

Proof Obligation 5. **Type of function** *Routing*

```
(defthm Vlstp-routing
  (let ((NodeSet (NodeSetGenerator pms)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp pms))
             (Vlstp (routing M NodeSet)))))
```

We have shown the main constraints on function *Routing*. Some local lemmas on the witness are necessary. There are two additional constraints. One that checks that function `Routing` outputs a true list. Another one checks that function `Routing` returns `nil` if the initial missive list is empty.

# 6. SCHEDULING POLICY

In the next subsection, we introduce the generic definition of the scheduling policy. Then, we give its associated proof obligations.

## 6.1 Generic Definition

Function *Scheduling* takes as arguments the travel list produced by function *Routing* and the list *att* of the attempt numbers at the nodes. It returns two travel lists: the list *Scheduled* and the list *Delayed*. It also updates the attempt number list *att*. The functionality of *Scheduling* is the following:

$$Scheduling : \mathcal{D}_{\mathcal{V}} \times AttLst \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{V}} \times AttLst \quad (7)$$

Its ACL2 signature is the following:

```
((scheduling * *) => (mv * * *))
```

For every scheduled travel of a missive that has several routes, the scheduling function generally keeps only one route. In order to avoid the introduction of a new data type, we consider scheduled travels like "classical" travels, *i.e.* travels that contain a list of routes, even if this list has only one element.

The local witness is very simple but fully fullfils its duty. If the sum of all attemps is zero, all travels are delayed. Otherwise, all travels are scheduled and each node with at least one attempt left consumes one attempt (function `consume-attempts`).

```
(local
  (defun scheduling (V att)
    ;; local witness
    (mv
    ;; scheduled frames
     (if (zp (SumOfAttempts att))
         nil  ;; no attempt left, no schedule
       V)  ;; otherwise all scheduled
    ;; delayed frames
     (if (zp (SumOfAttempts att))
         V ;; no attempt left, all delayed
       nil) ;; otherwise no delayed
     (if (zp (SumOfAttempts att))
         att ;; no attempt left, att unchanged
       (consume-attempts att))))) ;; consume att
```

## 6.2 Proof Obligations

First, if the list $\mathcal{V}$ is a valid travel list, the lists *Scheduled* and *Delayed* are also valid.

Proof Obligation 6. **Type of** *Scheduled* **and** *Delayed*.

```
(defthm Vlstp-scheduled-delayed
  (implies (Vlstp V)
    (and
     (Vlstp (mv-nth 0 (scheduling V att)))
     (Vlstp (mv-nth 1 (scheduling V att))))))
```

At each scheduling round, all travels of $\mathcal{V}$ are analyzed. If several travels are associated to a single node, this node consumes one attempt for the set of its travels. At each call to *Scheduling*, an attempt is consumed at each node. If all attemps have not been consumed, the sum of the remaining attempts after the application of function *Scheduling* is strictly less than the sum of the attempts before the application of *Scheduling*. This is expressed by the following proof obligation:

Proof Obligation 7. **Consume one attempt.**

```
(defthm consume-at-least-one-attempt
  (mv-let (Scheduled Delayed newatt)
          (scheduling V att)
          (declare (ignore Scheduled Delayed))
          (implies (not (zp (SumOfAttempts att)))
                   (< (SumOfAttempts newatt)
                      (SumOfAttempts att)))))
```

The delayed travels are converted to missives in the recursive call of $GeNoC$. This process should result in a sublist of the initial list of missives. To obtain a valid missive list, the

information contained in the delayed travels must be identical to the information contained in the initial list $\mathcal{V}$. The list of the delayed travels must be a sublist of $\mathcal{V}$. Formally, one ensures that list *Delayed* is equal to filtering the initial travel list according to the identifiers of *Delayed*. That corresponds to the following proof obligation:

PROOF OBLIGATION 8. **Correctness of** *Delayed*.

```
(defthm delayed-travel-correctness
  (mv-let
   (Scheduled Delayed newatt)
   (scheduling V att)
   (declare (ignore newatt scheduled))
   (implies (Vlstp V)
            (equal Delayed
                   (filter V
                           (v-ids Delayed)))))
  :rule-classes nil)
```

This rule is likely to introduce loops in the rewriter because *Delayed* appears in the left and the right hand side. Therefore, we do not store it as a rule.

Generally, the scheduling function only keeps one route for every scheduled travel. Consequently, the list *Scheduled* is not exactly a sublist of the initial travel list $\mathcal{V}$. The identifiers and the frames are not modified. We check that the route, or more generally, the routes of a scheduled travel belong to the routes associated with the corresponding initial travel.

Let us consider predicate `s-travel-correctness`. It takes as arguments two travel lists `sV` and `V/sids`. It checks that these lists have an equal length. It recursively checks that each element of `sV` has the same identifier, the same frame of the corresponding element in `V/sids`. It also recursively checks that routes of elements of `sV` are part of the routes of corresponding elements in `V/sids`. The definition of this predicate is the following:

```
(defun s-travel-correctness (sV V/sids)
   (if (endp sV)
       (if (endp V/sids)
           t
         nil)
     (let* ((str (car sV))
            (tr (car V/sids)))
       (and (equal (FrmV str) (FrmV tr))
            (equal (IdV str) (IdV tr))
            (subsetp (RoutesV str) (RoutesV tr))
            (s-travel-correctness (cdr sV)
                                  (cdr V/sids))
))))
```

The constraint regarding the scheduled travels states that this predicate must be satisfied if `sV` is the list of the scheduled travels and `V/sids` is the initial travel list filtered according to the identifiers of the scheduled travels.

PROOF OBLIGATION 9. **Correctness of** *Scheduled*.

```
(defthm scheduled-travels-correctness
    (mv-let (Scheduled Delayed newatt)
            (scheduling V att)
            (declare (ignore Delayed newatt))
            (implies (Vlstp V)
                     (s-travel-correctness
                      Scheduled
                      (filter V
                           (V-ids Scheduled))))))
```

Since routes of travels in *Scheduled* are routes of travels of $\mathcal{V}$, function *Scheduling* preserves the correctness of routes. We prove outside the `encapsulate` that the list *Scheduled* satisfies predicate `CorrectRoutesp`.

The goal of the scheduling policy is to partition a travel list into two exclusive lists: *Scheduled* and *Delayed*. The intersection of the identifiers of these two lists must be empty.

PROOF OBLIGATION 10. **Mutual Exclusion.**

```
(defthm not-in-delayed-scheduled
  (mv-let (scheduled delayed newatt)
          (scheduling V att)
          (declare (ignore newatt))
          (implies (Vlstp V)
                   (not-in (v-ids delayed)
                           (v-ids scheduled)))))
```

We have exposed the main constraints about function *Scheduling*. For technical reasons, additional constraints are necessary. To apply function `mv-nth` properly, function *Scheduling* needs to return a list of values. This property is not added by ACL2 from the signature. We also need to know that lists *Scheduled* and *Delayed* are true lists.

# 7. OVERALL MODEL

The definition of function *GeNoC* follows Figure 3. Function *ComputeMissives* applies function *send* to each transaction of the initial list $\mathcal{T}$. This produces the corresponding list of missives. Function *Routing* computes the routes of each missive and function *Scheduling* fixes the scheduled and the delayed travels. Function *ComputeResults* applies function *recv* to each scheduled travel to obtain results. Delayed travels are converted to missives. If all attempts have not been consumed, delayed travels are processed again from function *Routing*. Otherwise, delayed travels constitute the aborted communications.

The correctness of function *GeNoC* has been defined in section 2, with respect to results only. As explained in section 3, quantifiers are replaced by predicates on lists. In ACL2, the correctness of *GeNoC* concerns the results and the filtering of the initial transactions with the identifiers of the results. Thus, predicate `GeNoC-correctness` checks that each result corresponds to a transaction with the same identifier, message and destination. We obtain the following:

THEOREM 1. **ACL2 Correctness of** *GeNoC*.

```
(defthm GeNoC-is-correct
  (let ((NodeSet (NodeSetGenerator pms)))
    (mv-let (res abt)
            (GeNoC Trs NodeSet att)
            (declare (ignore abt))
            (implies (and (Tp Trs NodeSet)
                          (ValidParamsp pms))
                     (GeNoC-correctness
                      res
                      (filter Trs (R-ids res)))))))
```

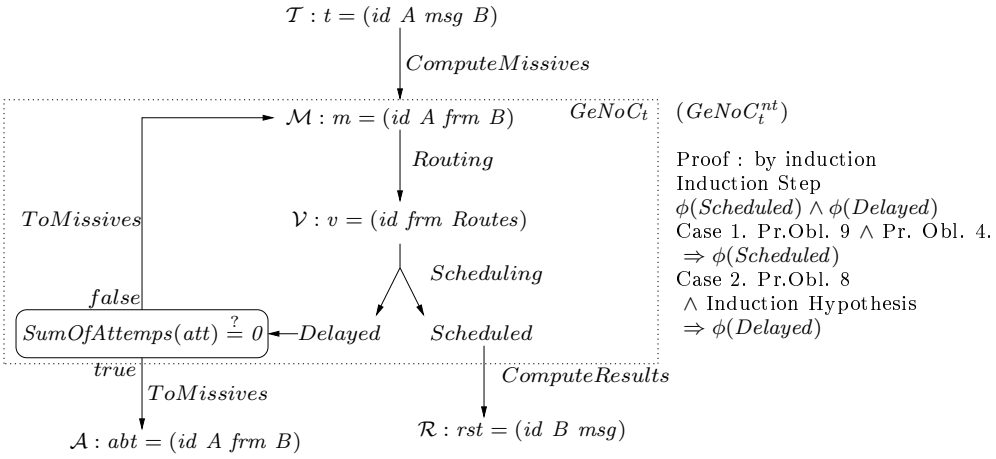


**Figure 3: Proof of *GeNoC***

This theorem is proven by induction on the structure of function *GeNoC*. The inductive proof only concerns the composition of functions *Routing* and *Scheduling*. Thanks to proof obligation 10, the scheduled and the delayed travels can be proven separately. Scheduled travels have a correspondance with the travel list input in *Scheduling* (proof obligation 9). Function *Routing* produces correct routes (proof obligation 4), which are still correct after *Scheduling*. So, frames and destinations after *Scheduling* match the missives input to function *Routing*. Results are matched to the initial transactions using the correctness of interfaces (proof obligation 1). The delayed travels are proven using the induction hypothesis and proof obligation 8.

The proof of *GeNoC* and its modules involves 71 functions, 119 theorems in 1864 lines of code. Only one fourth of these is dedicated to the encapsulation of the different modules. Most of the definitions and theorems concern data types and the proof of the overall correctness. This makes *GeNoC* "relatively simple" to use, because users will only be concerned with the modules. We import the last book on arithmetic developed by R. Krug and books on lists by B. Bevier.

# 8. METHODOLOGY AND CASE STUDIES

The generic model defines also a methodology for the specification and the validation of routing algorithms, scheduling policies and interfaces. In this section, we first give an overview of different concrete instances of *GeNoC*. As a case study, we apply *GeNoC* on an XY routing algorithm in a 2D mesh.

## 8.1 Overview

To show the adequacy between our generic model and real applications, we apply *GeNoC* to a litany of concrete designs. Any combination of these different concrete instances is defined and validated by generic function *GeNoC*, that means without any additional effort. These concrete instances are summarized in Fig 4.

We have shown that the circuit [10] and the packet [11] switching techniques are concrete instances of *Scheduling*. Based on previous work [9], we proved that bus arbitration in the AMBA AHB is also a valid instance of the generic scheduling policy. From Moore's work on asynchrony [5], we proved that his model of the biphase protocol constitutes a valid instance of the interfaces. We have modeled an Ethernet controler[2] and we are investigating its compliance with *GeNoC*.

In the next subsection, we illustrate our approach on an XY routing algorithm, with an ACL2 oriented presentation. This proof has already been presented to a general audience [11]. The routing in the Octagon network [4] - developed by STMicroelectronics - also constitutes a valid instance of our generic routing function. Finally, we are currently working on the proof that an adaptive routing algorithm - the double Y channel algorithm in a 2D mesh - is a valid instance of function *Routing*. More details about all these studies can be found in Schmaltz's thesis [8].

We now detail the methodology associated with the routing algorithm and illustrate it on an XY routing algorithm.

## 8.2 Case Study: XY routing

Regarding the routing function, the methodology proceeds in two steps. First, nodes and parameters are defined and proven compliant with the `encapsulate` given in section 4. Then the routing algorithm is modeled as a function that matches function *Routing*. In both steps, checking the compliance with the generic model is done by proving `t` as explained in section 3.1.

### 8.2.1 Mesh Node Definition

In a 2D mesh, a node is represented by a pair of coordinates on the X and Y axes. A pair of coordinates is recognized by predicate `Coordinatep`. A list of coordinates is recognized by predicate `mesh-nodesetp`.

Mesh parameters are the number of nodes in each dimension; they are recognized by predicate `ValidParamsp2D`. Let $N_X$ and $N_Y$ denote the number of nodes in the first and the second dimension. The node set, *i.e.* the set of coordinates from $(0, 0)$ to $((N_X - 1), (N_Y - 1))$, is generated by function `mesh-nsgen`. It is defined as follows.

Function $XGen(N_X, y)$ takes as arguments the number $N_X$ of nodes in the first dimension and a constant $y$ in the second dimension. It generates all admissible pairs for that

[2]This work has been done during a visit of the first author at the University of Texas at Austin, in cooperation with Warren Hunt.

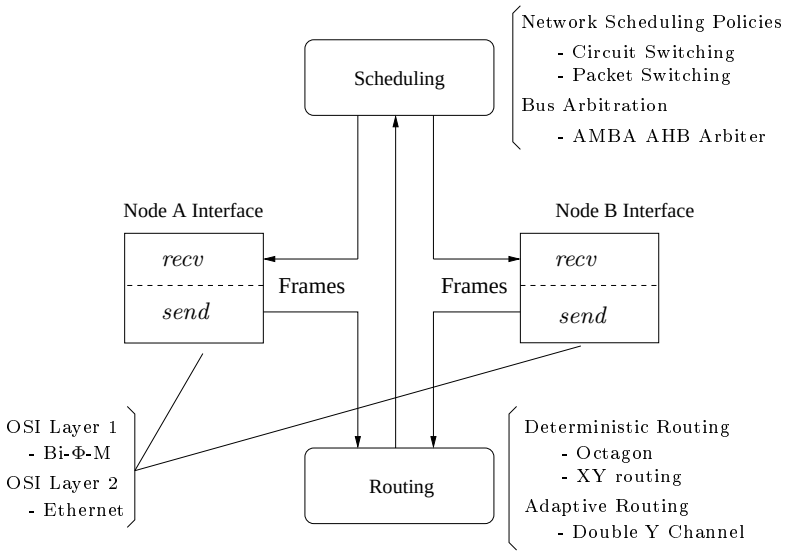


**Figure 4: Concrete Instances of *GeNoC***

particular $y$. Function `mesh-nsgen` computes the coordinates by applying function *XGen* to all values of $y$ ranging from zero to $N_Y - 1$. To prove the main constraint on the node definition, we first prove that the generation on the X axis is valid, and use this fact prove that nodes generated on the Y axis are valid.

THEOREM 2. **Mesh Nodes Validation.**

```
(defthm 2d-mesh-nodesetgenerator
  (implies (ValidParamsp2D pms)
           (mesh-nodesetp (mesh-nsgen pms))))
```

Once this theorem is proven, we check that the coordinates are a valid instance of the generic node definition by proving `t` as explained in section 3.1.

### 8.2.2 XY Routing Algorithm

Let $s = (s_x, s_y)$ be a node containing a packet addressed to node $d = (d_x, d_y)$. In the XY algorithm, the X direction has higher priority. If the X of destination $d$ is greater (resp. less) than the X of origin $s$, the next node is the node $(s_x + 1, s_y)$ (resp. $(s_x - 1, s_y)$ ) on the X-axis. Otherwise, the X's are equal and we compare the Y's: the next node is either $(s_x, s_y + 1)$ or $(s_x, s_y - 1)$ on the Y-axis. This algorithm is applied recursively to compute the route from a source to a destination. The measure is simply the sum of the absolute values of the difference of the coordinates.

DEFINITION 1. **XY Routing Algorithm**

```
(defun xy-routing (from to)
 (declare (xargs :measure (XY-measure from to)))
 ;; from = (x_o y_o) dest = (x_d y_d)
 (if (or (not (coordinatep from))
         (not (coordinatep to)))
     nil
   (let ((x_d (car to))
         (y_d (cadr to))
         (x_o (car from))
         (y_o (cadr from)))
     (if (and (equal x_d x_o)  ;; x_d = x_o
              (equal y_d y_o)) ;; y_d = y_o
         ;; if the destination is equal to
         ;; the current node, we stop
         (cons from nil)
       (if (not (equal x_d x_o))  ;; x_d /= x_o
           (if (< x_d x_o) ;; decreasing x
               (cons
                from
                (xy-routing (list (- x_o 1) y_o)
                            to))
             ;; x_d > x_o
             (cons
              from
              (xy-routing (list (+ x_o 1) y_o) to)))
             ;; otherwise we test the y-direction
             ;; y_d /= y and x_d = x_o
         (if (< y_d y_o)
             (cons
              from
              (xy-routing (list x_o (- y_o 1)) to))
           ;; y_d > y_o
           (cons
            from
            (xy-routing (list x_o (+ y_o 1)) to))))
))))
```

We then cast this function such that it matches the definition of *Routing*:

DEFINITION 2. **Matching** *Routing*.

```
(defun XYRouting (M NodeSet)
  (declare (ignore NodeSet))
  (xy-routing-top M))
```

*where:*

```
(defun xy-routing-top (M)
  (if (endp M)
      nil
    (let* ((miss (car M))
```

```
        (from (OrgM miss))
        (to (DestM miss))
        (id (IdM miss))
        (frm (FrmM miss)))
  (cons (list id frm
              (list (xy-routing from to)))
        (xy-routing-top (cdr M))))))
```

This function is a valid instance of the generic routing function of *GeNoC* if it computes a route that satisfies predicate `CorrectRoutesp`:

THEOREM 3. **Validity of the XY algorithm.**

```
(defthm CorrectRoutesp-XYRouting
  (let ((NodeSet2D (mesh-nsgen pms)))
    (implies (and (ValidParamsp2D pms)
                  (Missivesp M NodeSet2D))
             (CorrectRoutesp (xy-routing-top M)
                             M NodeSet2D))))
```

PROOF. Most properties defined in `CorrectRoutesp` are straightforward, and the ACL2 proofs are automatic. Only one proof requires an interaction with the prover: showing that each route uses valid nodes only. The set of nodes is generated by function `mesh-nsgen` and is made of all natural pairs $(x, y)$ such that $0 \leq x < N_X$ and $0 \leq y < N_Y$. The proof strategy is to show that any set of coordinates satisfying these inequalities is a subset of `NodeSet2D`. Then, it suffices to show that the route produced by function `xy-routing` satisfy these inequalities. The validation of this proof tactic requires 5 lemmas et 2 additional functions. The proof of the "closure" of `xy-routing` on `NodeSet2D` requires 30 lemmas. ACL2 needs a hint for only two of them. □

Before checking that this routing function is a valid instance of the generic routing function, we prove that it produces a valid travel list. This proof is obvious and not detailed further. Once again, the compliance of the XY routing algorithm with *GeNoC* is done by proving $t$.

## 9. CONCLUSIONS

We have presented the modeling of *GeNoC* in the ACL2 logic. We have shown how ACL2 can automatically produce proof obligations for particular instances of the generic model. We kept the number of encapsulated constraints as low as possible. Thus, the proof effort for particular instances is minimized.

The translation of our general theory in ACL2 is not direct. In higher order logics, predicates over functions would have replaced the encapsulations. Nevertheless, the functional instanciation principle automatically produces conjectures for particular applications. Moreover, ACL2 tries to prove them automatically. The user is directly left with the more interesting part of the proofs. The ACL2 implementation of *GeNoC* benefits greatly from these two principles.

On-going work at TIMA involves the application of *GeNoC* to wormhole routing, and the elaboration of a refinement method to derive the correctness of a particular hardware implementation.

## 10. ACKNOWLEDGMENTS

The authors would like to thank J Strother Moore, Matt Kaufmann and Warren Hunt for valuable remarks and helpful advice.

## 11. REFERENCES

[1] H. Amjad. Model Checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September 2004.

[2] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan-Kaufmann Publisher, 2004.

[3] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Rădulescu. Deadlock Prevention in the Æthereal protocol. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 345–348, 2005.

[4] K. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking Systems On Chip. *IEEE Micro*, pages 36–45, September-October 2002.

[5] J. S. Moore. A Formal Model of Asynchronous Communications and Its Use in Mechanically Verifying a Biphase Mark Protocol. *Formal Aspects of Computing*, 6(1):60–91, 1993.

[6] J. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. In *$34^{th}$ Design Automation Conference (DAC'96)*, pages 178–183, 1997.

[7] A. Roychoudhury, T. Mitra, and S. Karri. Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Design Automation and Test Europe (DATE'03)*, pages 828–833, 2003.

[8] J. Schmaltz. *Une formalisation fonctionnelle des communications sur la puce*. PhD thesis, Joseph Fourier University, Grenoble, France, January 2006. In French. A partial translation is available upon request to the first author.

[9] J. Schmaltz and D. Borrione. Verification of a Parameterized Bus Architecture Using ACL2. In *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications*, April 2003.

[10] J. Schmaltz and D. Borrione. A Functional Approach to the Formal Specification of Networks on Chip. In A. Hu and A. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 52–66, Austin, Tx, USA, November 2004. Springer-Verlag.

[11] J. Schmaltz and D. Borrione. A Generic Network on Chip Model. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 310–325, Oxford, UK, August 2005. Springer-Verlag.

[12] G. Spirakis. Beyond Verification: Formal Methods in Design. In A. Hu and A. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, Austin, Texas, USA, November 2004. Springer-Verlag. Invited Speaker.