Formalizing On Chip Communications in a Functional Style

Julien Schmaltz[1] and Dominique Borrione[2]

[1] Saarland University, Department of Computer Science
FR 6.2 Informatik, PostFach 151150, 66 041 Saarbrücken,Germany
julien@cs.uni-sb.de
[2] TIMA Laboratory - VDS Group
46, avenue Felix Viallet, 38031 Grenoble Cedex, France
Dominique.Borrione@imag.fr

Abstract. This paper presents a formal model for representing *any* on-chip communication architecture. This model is described mathematically by a function, named *GeNoC*. The correctness of *GeNoC* is expressed as a theorem, which states that messages emitted on the architecture reach their expected destination without modification of their content. The model identifies the key constituents common to *all* communication architectures and their essential properties, from which the proof of the *GeNoC* theorem is deduced. Each constituent is represented by a function which has no *explicit* definition but is constrained to satisfy the essential properties. Thus, the validation of a *particular* architecture is reduced to the proof that its concrete definition satisfies the essential properties. In practice, the model has been defined in the logic of the ACL2 theorem proving system. We define a methodology that yields a systematic approach to the validation of communication architectures at a high level of abstraction. To validate our approach, we exhibit several architectures that constitute concrete instances of the generic model *GeNoC*. Some of these applications come from industrial designs, such as the AMBA AHB bus or the Octagon network from ST Microelectronics.

1 Introduction

Current chip technology (65nm) allows the integration of several hundred million transistors on a single die, which requires a huge progress in design methodologies. Indeed, chip business is highly competitive and time to market has shrunk. A three month delay induces the loss of one fourth of the expected income [6]. To face this increasing time pressure, *systems on a chip* (SoC) are designed through a *platform* based approach: a new SoC is built according to a generic architecture, using pre-designed parameterized modules and processor cores. In that context, the interconnect structure becomes challenging both for design and verification [26].

Until recently, most of the verification effort was spent on the processing elements, and the literature specifically devoted to the embedded communication

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/700

architecture is relatively sparse. Bus architectures, and their protocols, have been the subject of the earlier works on that topic. Roychoudhury *et al.* use the SMV model checker to debug an academic implementation of the AMBA AHB protocol [19]. Their model is written at the register transfer level and without any parameter. Roychoudhury *et al.* detect a live lock scenario that was caused by the implementation of their arbiter rather than by the protocol itself. More recently, Amjad [2] used a model checker, implemented in the HOL theorem prover, to verify the AMBA APB and AHB protocols, and their composition in a single system. Using model checking, safety properties are verified on each protocol individually. The HOL tool is used to verify their composition. In this work also, the model is at a low level of abstraction, and without any parameter.

Networks on a chip (NoC) are a more recent design paradigm, and little work has been done about their formal verification outside straightforward model checking on fixed structures. A notable exception is the work of Gebremichael *et al.* [10], who recently specified the Æthereal protocol [11] of Philips in the PVS logic. The main property they verified is the absence of deadlock for an arbitrary number of masters and slaves.

At this point, it is worth noting that the above mentioned formal verification efforts, devoted to communication architectures and protocols, were performed at the register transfer level (RTL), on a very specific design. This level was considered appropriate when the same source was generating the synthesizable design for the full system. With the advent of outsource IP's and platform based design, the current trend in the SoC design community is to raise the level of abstraction [26] and rely on verified parameterized library modules. This requirement will soon extend to communication network kernels, yet a formal theory for this category of functional modules is non existing today. In effect, most textbook (*e.g.* [8]) *describe* architectures in an informal manner.

On the path to the definition of a formal theory of communications, two important studies have already treated part of it. Moore [17] defined a formal model of asynchrony by a function in the Boyer-Moore logic [5], and showed how to use this general model to verify a biphase mark protocol. More recently, Herzberg and Broy [12] presented a formal model of stacked communication protocols, in the sense of the OSI reference model. In a relational framework supporting a component-oriented view, they defined operators and conditions to navigate between protocol layers. Herzberg and Broy's framework considers all OSI layers. Thus, it is more general than Moore's work, which is targeted at the lowest layer. In contrast, Moore provides mechanized support. Both studies focus on protocols and do not consider the underlying interconnection structure explicitly.

The long term objective of our research is to support the validation of abstract specifications for on chip communication architectures, and the verification of their correct implementation by a given, possibly parameterized, IP. To this aim, we provide a general formal framework that encompasses the essential constituents of communication modules - *i.e.* protocols *and* topologies, routing algorithms and scheduling policies - and applies to a wide variety of communication architectures. It is essential that our theory be directly expressible in the logic of an interactive theorem prover, either first or higher order, to provide mechanized reasoning support.

This paper presents what constitutes, to our knowledge, a first proposal for a formal theory for communication architectures. Communications on the chip share many concepts with computer networks, but work on a different time scale. Systems on a chip often have very hard time, heat and power constraints. On chip communications must be predictable: losing and resending a message, or reordering message pieces is unacceptable within a SoC, while it is current practice on the Internet. On chip communications are more constrained, and their topology is statically defined, which simplifies the protocols. This paper only deals with these restricted communications systems: buses and NoC's. We formalize a generic communication architecture in functional form. A global function, called $GeNoC$, formalizes the interactions between the three key constituents: interfaces, routing and scheduling.

This generic model makes no assumption on the protocol, the topology, the routing algorithm, or the scheduling policy. To abstract from any particular architecture, we have identified essential properties (considered proof obligations or simply constraints) for each constituent. Those imply the overall correctness of $GeNoC$. Hence, the validation of any particular architecture is reduced to the proof that each one of its constituents satisfies the generic constraints. By embedding our theory in the logic of an automated proof assistant, we provide a tool to specify and to validate network on a chip description at a high level of abstraction. For any concrete architecture, the proof assistant automatically generates the proof obligations that must be satisfied to prove the compliance of this architecture with our theory.

This paper is structured as follows. The next section presents a motivating example network, and defines our notations. Section 3 gives an overview of our theory. Section 4 constitutes the core of the paper and our original contribution: it precisely defines the functions and proof obligations for the main constituents of a network on chip. Section 5 exposes our methodology for applying our model to a practical network on chip in a systematic way, and gives an overview of our experiments on a variety of communication architectures. The instanciation of the $GeNoC$ model to the "Octagon" design by STMicroelectronics is used as an illustration. Finally, section 6 concludes the paper and gives future research directions.

2 Background for the Theory

Our theory relies on some background principles and fundamental common features of all communication architectures. To make our theory easily expressible in interactive proof assistants, we define it using lists and their associated operators, as introduced at the end of this section. Let us first start with an example.

2.1 An NoC Example: the Octagon

This network on a chip has been designed by STMicroelectronics [14]. A basic Octagon unit consists of eight nodes and twelve bidirectional links (Figure 1). It has two main properties: the communication between any pair of nodes requires at most two hops, and it has a simple, shortest-path routing algorithm [14].

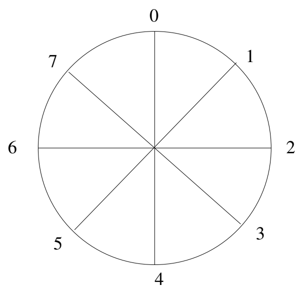

Fig. 1. Basic Octagon Unit

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 ($PackAd$) to its own address ($NodeAd$) to determine the next action. The node computes the relative address of a packet as:

$$RelAd = (PackAd - NodeAd) \bmod 8 \tag{1}$$

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

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

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

2.2 A Unifying Model

The previous example is generalized to the communication model of Figure 2. An arbitrary but finite number of *nodes* is connected to some communication architecture, bus or network. Topologies, routing algorithms and scheduling policies are its essential constituents. To distinguish between interface-application and interface-interface communications, an interface and an application communicate using *messages*; two interfaces communication using *frames*.

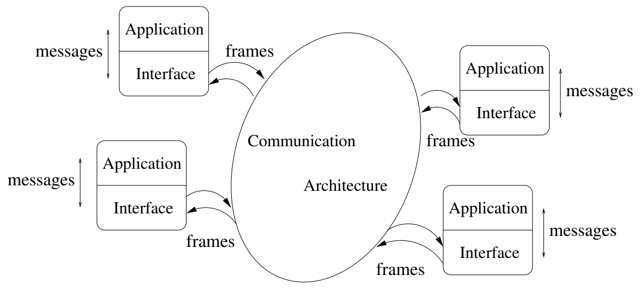

Fig. 2. Communication Model

Applications represent the computational and functional aspects of nodes. They are either active or passive. Typically, active applications are processors and passive applications are 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 requests. We focus on communications between distant nodes. We suppose that in every communication, the destination node is distinct from the source node.

2.3 Lists: Notations and Operators

Lists are essential to the implementation of our formalism. We briefly present the notations and the functions used to manipulate them. Notations about lists are summarized in Table 1.

Letters l or L are used to denote a list or a list of lists. List elements are often represented by letter e. The empty list is denoted by ϵ. A list l is a finite sequence of k values indexed from 0 to $k-1$, $l = (l[i])_{i\in[0;k-1]}$.

$Len(l)$ returns the length of list l (its number of elements), and $Last(l)$ returns its last element. Predicate $NoDuplicatesp(l)$ recognizes a list in which each two elements are distinct. The type of a list l_1 is defined by the membership of its elements to a given set E, and is denoted with the $\subseteq_l$ operator.

Adding an element e in front of a list l creates a new list l', noted $l' = e.l$. Element e takes index 0 in l'. Elements of l' with an index i greater that 0 are elements of l with index $i-1$. If the list is a list of lists, e is a list. The append of two lists, l_1 and l_2, of the same type is denoted $l_1 \sqcup l_2$, resulting in a list of

Name	Purpose
$e.l$	add element e to list l
$l_1 \subseteq_l E$	l_1 is a list of E (type)
$l_1 \sqcup l_2$	append of l_1 and l_2
$e \in_l l_1$	e is an element of list l_1
$l_1 \sqsubseteq l_2$	l_1 is included in l_2
$l_1 \sqcap l_2$	elements common to l_1 and l_2
$List(l_1, l_2)$	juxtaposition of lists l_1 and l_2
$Len(l)$	the number of elements contained in l
$Last(l)$	the last element of l
$NoDuplicatesp(l)$	recognizes a list l with no duplicate
ϵ	the empty list
$l[i]$	an element of list l, $0 \leq i \leq Len(l) - 1$

Table 1. Notations and functions used to manipulate lists

this type. If the lists have not the same type, their juxtaposition is obtained by function $List(l_1, l_2)$. An element e is an element of a list l if and only if e is a value of l. $e \in_l l_1$ reads: e is an element of list l_1. A list l_1 is *included* in a list l_2 , denoted $l_1 \sqsubseteq l_2$, if and only if every element of l_1 is an element of l_2. The empty list, ϵ, is included in all lists. Examples: the list $(1\ 1\ 1)$ is included in the list (1); the list $(3\ 2)$ is included in the list $(1\ 2\ 3)$. The list l in which the first occurrence of an element e has been removed is noted $l \setminus e$. The list l' containing all the elements that are elements of lists l_1 and l_2 is noted $l' = l_1 \sqcap l_2$. This list preserves the element ordering of l_1. For instance, $(1\ 2\ 5\ 3) \sqcap (1\ 2\ 1\ 3\ 4) = (1\ 2\ 3)$. The definition of operator $\sqcap$ is as follows:

$$l_1 \sqcap l_2 \triangleq \begin{cases} \epsilon & \textbf{if } l_1 = \epsilon \vee l_2 = \epsilon \\ l'_1 \sqcap l_2 & \textbf{if } l_1 = e.l'_1 \wedge e \notin_l l_2 \\ e.(l'_1 \sqcap (l_2 \setminus e)) & \textbf{if } l_1 = e.l'_1 \wedge e \in_l l_2 \end{cases} \tag{2}$$

If the elements e of a list L are lists, the list of the elements of L with the same index i in each e is noted $L_{\lfloor i}$.

In our model, the meaning of the elements of e is often given by an identifier. For readability, we shall use the identifier rather than its index. For instance, assume that e is a list composed of a key, a name and a surname: $e = (key\ name\ surname)$. Let L be a list of elements e of this kind. The list of the keys is noted $L_{\lfloor key}$, the list of the names $L_{\lfloor name}$ and the list of the surnames $L_{\lfloor surname}$.

Very often, a list is built by the application of a function f to every element of a list l. This operation corresponds to a higher-order function φ that takes as arguments a function f and a list l. Function φ returns the list of the results of the application of f to every element of l [1]. As function f could be complex, it is not always practical to have it explicitly formulated. Often, it suffices to express the modification done on each element. To alleviate the notation, the

[1] In functional programming, this corresponds to the map operation

application of function φ is noted using operator Λ defined as follows:

$$\bigwedge_{e \in_l l} f(e) \equiv \varphi(l, f) \triangleq \begin{cases} \epsilon & \textbf{if}\, l = \epsilon \\ f(e).\varphi(l', f) & \textbf{otherwise}\ l = e.l' \end{cases} \tag{3}$$

For instance, let l be a list of integer couples $e = (x_1\ x_2)$. The list l' of the sums $x_1 + x_2$ over the elements e of l is easily defined with operator Λ:

$$l' = \bigwedge_{e \in l} (e[0] + e[1])$$

3 Model Overview

3.1 Principles of *GeNoC*

Function *GeNoC* represents the transmission of messages on a generic communication architecture, with an arbitrary topology, routing algorithm and switching technique. 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: *send* encapsulates a message into a frame and injects the frame on the network; *recv* decodes the 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 functions *recv* and *send* ($recv \circ send$) is the identity function. Note that this property is also present in Moore's model of asynchrony, as well as in Herzberg and Broy's framework.
2. *Routing and topology* are represented by function *Routing*. The routing algorithm consists of the successive application of unitary moves. For each pair made of a source s and a destination d, *Routing* computes *all* the possible routes allowed by the unitary moves. The main constraint associated to *Routing* is that each route from s to d effectively starts in s and uses only existing nodes to end in d.
3. *The switching technique* is represented by function *Scheduling*. The scheduling policy participates in the management of conflicts, and computes a set of possible simulatenous communications. Formally, these commutations satisfy an *invariant.* Scheduling a communication, *i.e.* adding it to the current set of authorized communications, must preserve the invariant, at all times and in any admissible state of the network. The invariant is specific to the scheduling policy. In our formalization, the existence of this invariant is assumed but not explicitly represented. From a list of requested communications, function *Scheduling* extracts a sub-list of communications that satisfy the invariant. The rest represents the delayed communications

We stress the fact that all these functions are generic: their essential properties, called *proof obligations* or simply *constraints*, are formalized, but not their explicit definition.

3.2 Unfolding Function *GeNoC*

Function $GeNoC$ is pictured on Fig. 3. 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.

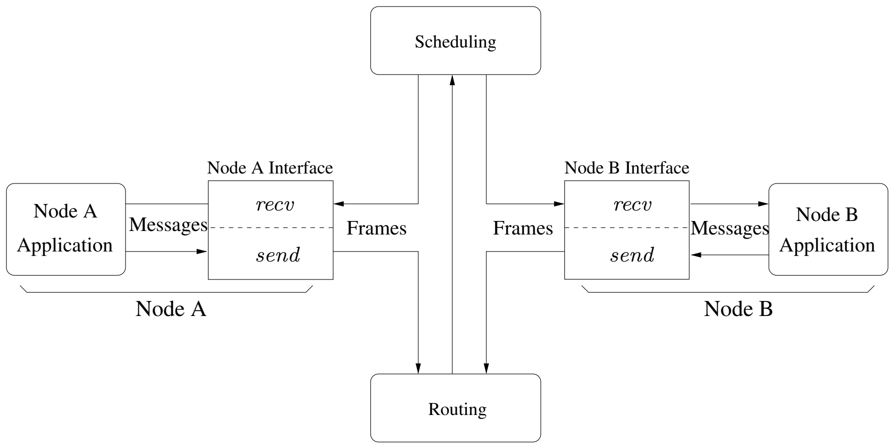

Fig. 3. $GeNoC$: A Generic Network

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 $\mathcal{T}_{lstp}(\mathcal{T}, NodeSet)$.

The unfolding of function $GeNoC$ is depicted in Figure 4. For every message in the initial list of transactions, function $ComputeMissives$ applies function $send$ to compute the corresponding frame. 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 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, 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 $\mathcal{V}_{lstp}(\mathcal{V})$. Function $Scheduling$ separates $\mathcal{V}$ into a list $Scheduled$ of scheduled travels and a list $Delayed$ of delayed travels. The results of the scheduled travels are computed by calling $recv$. 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 a *finite* number of attempts to every node. At every recursive call of $GeNoC$, every node with a pending transaction consumes one attempt. The ***association list** att* stores the attempts and $att[i]$ denotes the number of remaining attempts for the node i. Function $SumOfAtt(att)$ computes the sum of the remaining attempts for all the nodes and is used as the decreasing measure of parameter att. Function $GeNoC$ halts if all attempts have 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.

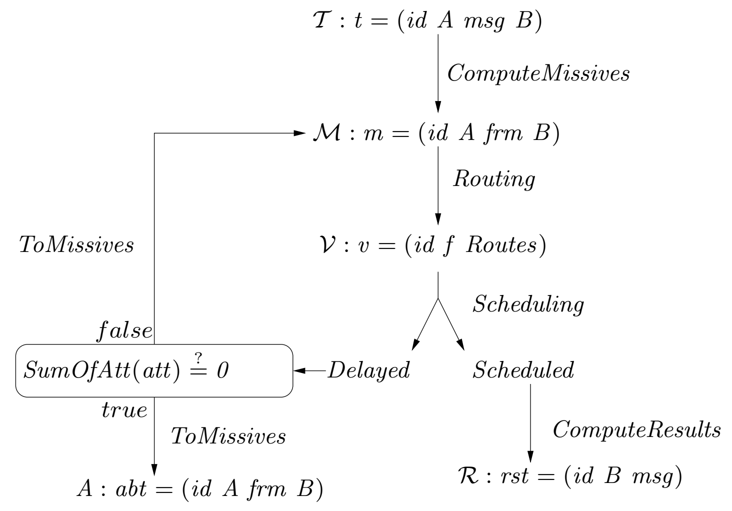
$\mathcal{T} : t = (id\ A\ msg\ B)$
ComputeMissives
$\mathcal{M} : m = (id\ A\ frm\ B)$
Routing
ToMissives
$\mathcal{V} : v = (id\ f\ Routes)$
Scheduling
false
$SumOfAtt(att) \overset{?}{=} 0$
Delayed
Scheduled
true
ToMissives
ComputeResults
$\mathcal{R} : rst = (id\ B\ msg)$
$A : abt = (id\ A\ frm\ B)$

Fig. 4. Unfolding of function $GeNoC$

The correctness of $GeNoC$ is expressed by two properties. First, the messages that are received are identical to the messages that were sent. Second, each message is received by its expected destination. Formally, this is expressed by the formula below, which shows that each result rst is obtained from a unique transaction t that has the same identifier, the same message and the same destination as rst.

$$\forall rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \begin{cases} \quad Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge\ Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge\ Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{cases} \tag{4}$$

4 Details of the Functional Model

4.1 Nodes and Parameters

Nodes are defined on an arbitrary domain, *GenNodeSet*, with characteristic function *ValidNodep*:

$$\forall x, \mathit{ValidNodep}(x) \Leftrightarrow x \in \mathit{GenNodeSet} \tag{5}$$

The set of nodes of a particular network is noted *NodeSet*. In all this section, we shall use a subscripted curly $\mathcal{D}$ to represent a domain of elements. For instance, $\mathcal{D}_{msg}$ is the domain of messages, $\mathcal{D}_{frm}$ is the domain of frames, etc.

4.2 Interfaces

Function *send* builds a *frame* from a *message* and function *recv* builds a *message* from a *frame*. Their functionality is:

$$\mathit{send} : \mathcal{D}_{msg} \rightarrow \mathcal{D}_{frm} \tag{6}$$

$$\mathit{recv} : \mathcal{D}_{frm} \rightarrow \mathcal{D}_{msg} \tag{7}$$

The constraint on these functions is that their composition is the identity function. The following proof obligation has to be relieved:

Proof Obligation 1 Validity of The Interface Functions.

$$\forall \mathit{msg} \in \mathcal{D}_{msg}, \mathit{recv} \circ \mathit{send}(\mathit{msg}) = \mathit{msg}$$

4.3 Routing

Principles and correctness criteria Let d be the destination of a frame standing at node s. In the case of deterministic algorithms, the routing logic of a network selects a unique node as the next step in the route from s to d. This logic is represented by function $\mathcal{L}(s, d)$. The list of the visited nodes for every travel from s to d is obtained by the successive applications of function $\mathcal{L}$ until the destination is reached, i.e. while $\mathcal{L}(s, d) \neq d$. The route from s to d is:

$$s, \mathcal{L}(s, d), \mathcal{L}(\mathcal{L}(s, d), d), \mathcal{L}(\mathcal{L}(\mathcal{L}(s, d), d), d), \ldots, d$$

A route is computed by function ρ_{det} that recursively applies function $\mathcal{L}$ from the source node to the destination node. Function ρ_{det} is defined as follows:

$$\rho_{det}(s, d) \triangleq \begin{cases} d & \textbf{if } s = d \\ s.\rho_{det}(\mathcal{L}(s, d), d) & \textbf{otherwise} \end{cases} \tag{8}$$

In the adaptive case, the routing logic offers at each intermediate node several "next" nodes. Several routes are possible between a source s and a destination

d. In that case, the routing algorithm is represented by function ρ_{ndet}, which computes *all* possible routes between nodes s and d.

To cover the general case, the routing algorithm is represented by function ρ, which takes as arguments a source node s and a destination node d. This function returns the list of the possible routes between s and d. Its functionality is the following, where $\mathcal{C}$ denotes a list of lists of nodes:

$$\rho : GenNodeSet \times GenNodeSet \rightarrow \mathcal{C} \tag{9}$$

Routing Termination Since function ρ is recursive, it must be shown to terminate, both to ensure the liveness of the network, and to be accepted by a proof assistant.

Let S be a set and $\prec_S$ be a total ordering relation on S. We recall that $(S, \prec_s)$ is a well-founded structure if any subset of S has a minimal element for $\prec_S$. Typically, the proof of termination of a function is done by showing that some *measure* on its parameters is decreasing on a well-founded structure for every recursive call of that function.

Let us return to the deterministic case and function ρ_{det}. Let $(S, \prec_S)$ be a well-founded structure (most often S is the set of naturals), and mes be a measure on S.

$$mes : GenNodeSet \times GenNodeSet \rightarrow S$$

To prove that ρ_{det} terminates, one needs to prove that the "governing" condition for the recursive call, namely $s \neq d$, implies that mes is decreasing. The following proof obligation has to be satisfied:

Proof Obligation 2 Termination Condition for ρ_{det}.

$$\forall s, d \in GenNodeSet, \exists mes : GenNodeSet \times GenNodeSet \rightarrow S,$$
$$s \neq d \Rightarrow mes(\mathcal{L}(s,d), d) \prec_S mes(s,d)$$

Routing Correctness The correctness of a route is defined according to a missive. A route r is correct with respect to a missive m if r starts with the origin of m, ends with the destination of m and every node of r belongs to the set of nodes of the network. Every correct route has at least two nodes. The following predicate defines these conditions:

Definition 1. ValidRoutep.

$$ValidRoutep(r, m, NodeSet) \triangleq \begin{cases} \quad r[0] = Org_{\mathcal{M}}(m) \\ \wedge\ Last(r) = Dest_{\mathcal{M}}(m) \\ \wedge\ r \subseteq_l NodeSet \wedge Len(r) \geq 2 \end{cases}$$

Whether routing is deterministic or adaptive, this predicate must be satisfied by all routes produced by function ρ. The following proof obligation has to be relieved:

Proof Obligation 3 Correctness of routes produced by ρ.

$$\begin{aligned}&\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet)\\ &\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet)\end{aligned}$$

Definition and Validation of Function *Routing* Function *Routing* takes as arguments a missive list and the set *NodeSet* of nodes of the network. It returns a travel list in which a list of routes is associated to each missive. The functionality of *Routing* is the following:

$$Routing : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \rightarrow \mathcal{D}_{\mathcal{V}} \tag{10}$$

Function *Routing* builds a travel list from the identifier, the frame, the origin and the destination of missives.

Definition 2. Function Routing
$Routing(\mathcal{M}, NodeSet) \triangleq$

$$\bigwedge_{m \in_l \mathcal{M}} List(Id_{\mathcal{M}}(m), Frm_{\mathcal{M}}(m), \rho(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)))$$

Concerning data types, one has to prove that function *Routing* produces a valid travel list if the initial missive list is valid.

Proof Obligation 4 Type of *Routing*.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(Routing(\mathcal{M}, NodeSet))$$

The definition of function *Routing* preserves the properties proved about the previous function ρ. Function *Routing* terminates and the routes of every travel satisfy predicate *ValidRoutep*. In a missive list, identifiers are unique. For every travel v produced by function *Routing*, there is a unique missive m such that its identifier equals the identifier of v and the frame of v equals the frame of m.

Theorem 1. Missive/Travel Match.

$$\begin{aligned}&\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow\\ &\forall v \in_l Routing(\mathcal{M}, NodeSet), \exists! m \in_l \mathcal{M}, \begin{cases} \quad Id_{\mathcal{V}}(v) = Id_{\mathcal{M}}(m)\\ \wedge\ Frm_{\mathcal{V}}(v) = Frm_{\mathcal{M}}(m)\end{cases}\end{aligned}$$

Proof. By definition of *Routing*.

Travels delayed by the scheduling function - but produced by function *Routing* - are converted back to missives by function *ToMissives*. The latter builds missives in the following manner. It takes the identifier and the frame of a travel. The origin and the destination of a missive are the first and the last node of a route. Function *ToMissives* is the reverse of function *Routing*.

Theorem 2. Routing ToMissives.

$$\forall \mathcal{M}, \mathcal{M}_{lstp} \Rightarrow ToMissives \circ Routing(\mathcal{M}, NodeSet) = \mathcal{M}$$

Proof. Frames are not modified by function *Routing*. Since the latter satisfies predicate *ValidRoutep* for all routes of all travels that it produces, the first and the last node of any route are equal to the origin and the destination of the initial missive.

4.4 Scheduling

Function *Scheduling* takes as arguments the travel list produced by function *Routing* and the list *att* of the remaining number of attempts. It updates *att* and returns two travel lists: the list *Scheduled* and the list *Delayed*. The functionality of *Scheduling* is:

$$Scheduling : \mathcal{D}_{\mathcal{V}} \times AttLst \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{V}} \times AttLst \qquad (11)$$

A scheduled travel only keeps one of the possible routes for the missive. For technical reasons, we avoid the introduction of a new data type and do not make a special case of scheduled travels: they contain a list of routes, even if this list has only one element.

The validation of *Scheduling* requires the satisfaction of several proof obligations.

In the following, the projection of a vector on one of its dimensions is denoted π_i^j, with the following functionality:

$$\pi_i^j : \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_j \rightarrow \mathcal{D}_i \qquad (12)$$

For instance, $\pi_1^2(x_1, x_2) = x_1$ and $\pi_2^2(x_1, x_2) = x_2$.

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

Proof Obligation 5 Type of *Scheduled* **and** *Delayed***.**
Let Scheduled be $\pi_1^3 \circ Scheduling(\mathcal{V}, att)$ *and*
Delayed be $\pi_2^3 \circ Scheduling(\mathcal{V}, att)$, *then* :

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \mathcal{V}_{lstp}(Scheduled) \wedge \mathcal{V}_{lstp}(Delayed)$$

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 attempts 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 6 Function *Scheduling* **consumes at least one attempt.**
Let natt be $\pi_3^3 \circ Scheduling(\mathcal{V}, att)$, *then:*

$$\begin{aligned} & SumOfAtt(att) \neq 0 \\ \rightarrow\ & SumOfAtt(natt) < SumOfAtt(att) \end{aligned}$$

The list of the delayed travels must be a sublist of $\mathcal{V}$. Formally, one ensures that for every delayed travel dtr, there exists a unique initial travel v such that dtr and v have the same identifier, the same frame and the same routes. Hence the following proof obligation:

Proof Obligation 7 Correctness of the delayed travels.
Let Delayed be $\pi_2^3 \circ Scheduling(\mathcal{V}, att)$*, then:*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall dtr \in_l Delayed, \exists ! v \in_l \mathcal{V}, \begin{cases} \quad Id_{\mathcal{V}}(dtr) = Id_{\mathcal{V}}(v) \\ \wedge\ Frm_{\mathcal{V}}(dtr) = Frm_{\mathcal{V}}(v) \\ \wedge\ Routes_{\mathcal{V}}(dtr) = Routes_{\mathcal{V}}(v) \end{cases}$$

Since the scheduling function only keeps one route for every scheduled travel, 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 of the corresponding initial travel. Formally, we ensure that for every scheduled travel str, there exists a unique initial travel v such that str and v have the same identifier, the same frame and that the routes associated with str are among the routes associated with v.

Proof Obligation 8 Correctness of the scheduled travels.
Let Scheduled be $\pi_1^3 \circ Scheduling(\mathcal{V}, att)$*, then:*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall str \in_l Scheduled, \exists ! v \in_l \mathcal{V}, \begin{cases} \quad Id_{\mathcal{V}}(str) = Id_{\mathcal{V}}(v) \\ \wedge\ Frm_{\mathcal{V}}(str) = Frm_{\mathcal{V}}(v) \\ \wedge\ Routes_{\mathcal{V}}(str) \sqsubseteq Routes_{\mathcal{V}}(v) \end{cases}$$

Since routes of travels in *Scheduled* are routes of travels of $\mathcal{V}$, function *Scheduling* preserves the correctness of routes. If routes of $\mathcal{V}$ satisfy predicate *ValidRoutep*, so do the routes of *Scheduled*.

A travel cannot, at the same time, be scheduled and delayed.

Proof Obligation 9 Mutual exclusion between *Delayed* and *Scheduled*.
Let Scheduled be $\pi_1^3 \circ Scheduling(\mathcal{V}, att)$ *and*
Delayed be $\pi_2^3 \circ Scheduling(\mathcal{V}, att)$*, then :*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow Delayed_{\lfloor id} \sqcap Scheduled_{\lfloor id} = \epsilon$$

4.5 Definition and Validation of *GeNoC*

The definition of function *GeNoC* and its correctness proof are summarized in Fig. 5. The recursive call in *GeNoC* only involves functions *Routing* and *Scheduling*. We define function $GeNoC_t$ to be the subfunction computing this recursion. It takes as arguments a list $\mathcal{M}$ of missives, the set *NodeSet* of nodes of the network, the list *att* of the attempt numbers and a travel list $\mathcal{V}$ that is initially empty. It returns two lists: a travel list that contains the frames received

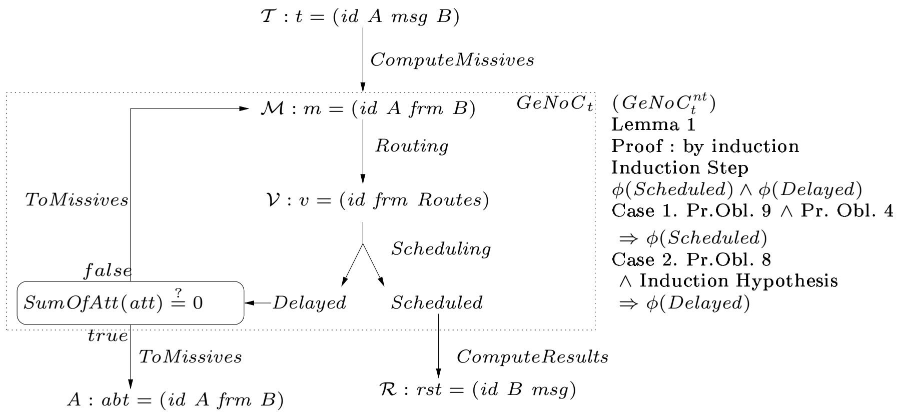
$\mathcal{T} : t = (id\ A\ msg\ B)$
ComputeMissives
$\mathcal{M} : m = (id\ A\ frm\ B)$
$GeNoC_t$
$(GeNoC_t^{nt})$
Lemma 1
Proof : by induction
Induction Step
$\phi(Scheduled) \wedge \phi(Delayed)$
Case 1. Pr.Obl. 9 $\wedge$ Pr. Obl. 4
$\Rightarrow \phi(Scheduled)$
Case 2. Pr.Obl. 8
$\wedge$ Induction Hypothesis
$\Rightarrow \phi(Delayed)$
Routing
ToMissives
$\mathcal{V} : v = (id\ frm\ Routes)$
Scheduling
false
$SumOfAtt(att) \overset{?}{=} 0$
Delayed
Scheduled
true
ToMissives
ComputeResults
$A : abt = (id\ A\ frm\ B)$
$\mathcal{R} : rst = (id\ B\ msg)$

Fig. 5. Proof of *GeNoC*

by the destination nodes of the missives in $\mathcal{M}$ and a list that contains the aborted missives. Its functionality is the following:

$$GeNoC_t : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \times AttLst \times D_{\mathcal{V}} \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{M}} \tag{13}$$

If all attempts have been consumed, $GeNoC_t$ returns the travels accumulated in $\mathcal{V}$ and the list of the remaining missives, *i.e.* the aborted missives. Otherwise, the travels produced by function *Routing* are passed to function *Scheduling*. The scheduled travels are added to the list $\mathcal{V}$. The delayed travels are converted to missives and constitute an argument of the recursive call to $GeNoC_t$. The remaining arguments are the updates of the lists *att* and $\mathcal{V}$.

Definition 3. Definition of $GeNoC_t$.

```
$GeNoC_t(\mathcal{M}, NodeSet, att, \mathcal{V}) \triangleq$
**if** $SumOfAtt(att) = 0$ **then**
 $List(\mathcal{V}, \mathcal{M})$
**else**
 **Let**$(ScheduledRtg\ DelayedRtg\ att_1)$ **be**
     $Scheduling(Routing(\mathcal{M}, NodeSet), att)$ **in**
       $GeNoC_t(ToMissives(DelayedRtg), NodeSet, att_1, ScheduledRtg \sqcup \mathcal{V})$
**endif**
```

The correctness of function $GeNoC_t$ is obtained if for every element *ctr* of the completed travels G, the frame and the last node of the route [2] of *ctr* are equal to the frame and the destination of the missive m in $\mathcal{M}$ that has the same identifier as *ctr*. This is expressed by the lemma below:

[2] Note that to keep our notations consistent, a travel is always made of a list of routes, even if this list has only one element.

Lemma 1. Correctness of $GeNoC_t$.

$$\forall ctr \in_l G, \exists! m \in_l \mathcal{M}, \begin{cases} \quad Id_{\mathcal{V}}(ctr) = Id_{\mathcal{M}}(m) \\ \wedge\ Frm_{\mathcal{V}}(ctr) = Frm_{\mathcal{M}}(m) \\ \wedge\ \forall r \in_l Routes_{\mathcal{V}}(ctr), Last(r) = Dest_{\mathcal{M}}(m) \end{cases}$$

Where:

$$G = \pi_1^2 \circ GeNoC_t(\mathcal{M}, NodeSet, att, \epsilon)$$

Proof. This theorem is proven by induction on the structure of function $GeNoC_t$. It follows from proof obligation 9 that the scheduled and the delayed travels can be proven separately. Scheduled travels have a correspondance with the travel list input in *Scheduling* (proof obligation 8). Function *Routing* produces correct routes (proof obligation 3), which are still correct after *Scheduling*. So, frames and destinations after *Scheduling* match the missives input to function *Routing*. The delayed travels are proven using the induction hypothesis and proof obligation 7.

Function *GeNoC* takes as arguments a list $\mathcal{T}$ of transactions, the set *NodeSet* of nodes of the network, the list *att* of attempt numbers. It returns the list $\mathcal{R}$ containing the results and the list A containing the aborted missives. It has the following functionality:

$$GeNoC : \mathcal{D_T} \times \mathcal{P}(GenNodeSet) \times AttLst \rightarrow \mathcal{D_R} \times \mathcal{D_M} \tag{14}$$

Function *ComputeMissives* applies function *send* to the message of each transaction of the list $\mathcal{T}$. This function produces a list of missives from the initial transactions. Its functionality is the following:

$$ComputeMissives : \mathcal{D_T} \rightarrow \mathcal{D_M} \tag{15}$$

It is defined as follows:

Definition 4. ComputeMissives.
$ComputeMissives(\mathcal{T}) \triangleq$

$$\bigwedge_{t \in_l \mathcal{T}} List(Id_{\mathcal{T}}(t), Org_{\mathcal{T}}(t), send(Msg_{\mathcal{T}}(t)), Dest_{\mathcal{T}}(t))$$

Function *ComputeResults* applies function *recv* to each frame of a travel list to produce a list of results. Its functionality is the following:

$$ComputeResults : \mathcal{D_V} \rightarrow \mathcal{D_R} \tag{16}$$

It is defined as follows:

Definition 5. ComputeResults.
$ComputeResults(\mathcal{V}) \triangleq$

$$\bigwedge_{tr \in_l \mathcal{V}} List(Id_{\mathcal{V}}(tr), Last(Routes_{\mathcal{V}}(tr)), recv(Frm_{\mathcal{V}}(tr)))$$

Function $GeNoC$ is defined using these functions and $GeNoC_t$. Function $ComputeMissives$ gives the first argument of $GeNoC_t$ from the transaction list $\mathcal{T}$. The last argument of $GeNoC_t$ is the empty list. The aborted missives are produced by function $GeNoC_t$. The definition of $GeNoC$ is the following:

Definition 6. Definition of $\boldsymbol{GeNoC}$**.**
$GeNoC(\mathcal{T}, NodeSet, att) \triangleq$
Let $(Responses\ Aborted)$ **be**
$GeNoC_t(ComputeMissives(\mathcal{T}), NodeSet, att, \epsilon)$ **in**
$List(ComputeResults(Responses), Aborted)$

The correctness of $GeNoC$ is defined by expression 4 defined in section 3.

Theorem 3. Correctness of $\boldsymbol{GeNoC}$**.**
Let $\mathcal{R}$ *be* $\pi_1^2 \circ GeNoC(\mathcal{T}, NodeSet, att)$ *in*

$$\forall rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \begin{cases} \quad Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge\ Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge\ Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{cases}$$

Proof. The last term of the conjunct is directly obtained from Lemma 1. From this lemma, it also follows that the frames produced by function $ComputeMissives$ are identical to the frames converted in messages by function $ComputeResults$. From proof obligation 1 on the interfaces, it comes that messages of results are equal to messages in the initial transaction list.

5 Methodology and Case Studies

We have embedded our theory in the logic of the ACL2 theorem proving system [15]. Despite the fact that ACL2 is first order, and does not support the explicit use of quantifiers, the choice of this system offered a number of advantages:

– The input language being a subset of Common Lisp, the functions are executable. It is realistic to execute a model on test benches, and visualize the behavior of a particular network specification, as a first debugging step before proceeding with human time consuming proofs. This feature is important also for quick software prototyping, as a basis of discussion with network designers.
– A large number of existing previous works are publicly available, and developing a new theory benefits from many layers of expert developments that extend the system first principles. Libraries of functions definitions and proven theorems can be compiled and stored for later use, restoring an environment is a single statement.
– Very powerful definition mechanisms, such as the *encapsulation principle*, allow to extend the logic and reason on undefined functions that satisfy one or more theorems, provided one witness can be exhibited. We made an

extensive use of this principle to prove the correctness of $GeNoC$ assuming the satisfaction of the constraints on the functions that formalize the network constituents.

– The combined use of typing predicates, list filtering, implication and recursive function definitions over list arguments provides a means to express universally quantified properties over domains, and the statement "there exists a unique element such that".

Applying a systematic, and reusable, mode of expression (see [25] for details), the complete $GeNoC$ formalization could be performed in the ACL2 logic, with the above listed benefits, and we thus benefited from the high degree of automated mechanized reasoning in ACL2.

The proof of the main theorem about $GeNoC$ and its modules involve 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, as we shall now discuss.

5.1 Overview of the Applications

In Fig 6, we summarize concrete instances of $GeNoC$. Any combination of these different concrete instances is defined and validated by generic function $GeNoC$, that means without any additional effort.

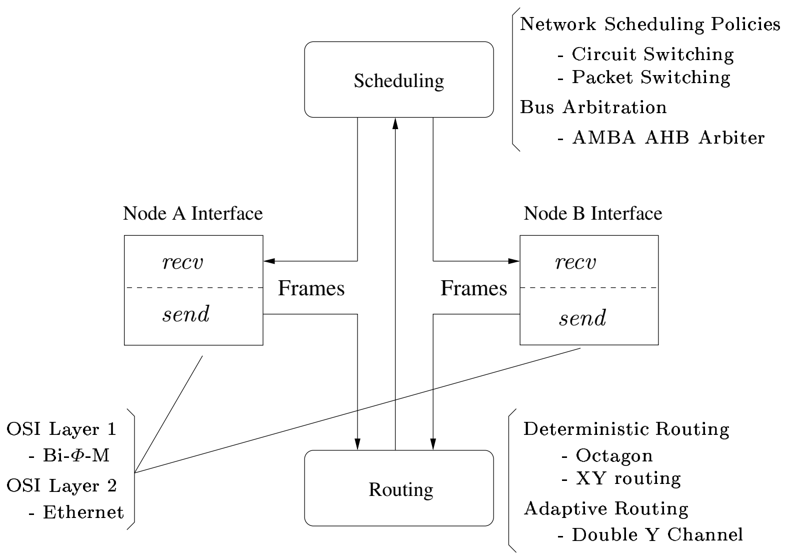

Fig. 6. Concrete Instances of $GeNoC$

We have shown that the circuit [23] and the packet [24] switching techniques are concrete instances of *Scheduling*. Based on previous work [22], 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 [17], we proved that his model of the biphase protocol constitutes a valid instance of the interfaces. We have modeled an Ethernet controler[3] and we are investigating its compliance with *GeNoC*.

In the next subsections, we illustrate our approach on the Octagon network. We first detail the methodology associated with the routing algorithm. Then, we apply it to the routing algorithm of the Octagon. A model and a proof of this network have already been presented [23], but with a different methodology. We have also shown that XY routing in a 2D mesh is also a valid instance of our generic model [24]. 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 [21].

5.2 Concrete instances of function *Routing*

The topology of a network determines the node numbering and the unitary moves allowed between two adjacent nodes. The routing function is defined by the successive applications of these moves. Before defining a particular routing function, one has to define the set of nodes.

Node Definition. Before all, one has to define the node definition domain, that is a particular instance of predicate $ValidNodep$, noted $ValidNodep_\sharp$. The generic definition domain $GenNodeSet$ becomes a particular domain $GenNodeSet_\sharp$, the naturals for instance. One has to give a concrete definition of Equation 5, that is:

$$\forall x,\ ValidNodep_\sharp(x) \Leftrightarrow x \in GenNodeSet_\sharp \tag{17}$$

Routing Definition First, we identify the moves allowed between two adjacent nodes. As we consider regular network (or a regularization of an irregular network), these moves are all identical at each point of the network. Identifying these unitary moves defines a concrete instance, $\mathcal{L}_\sharp$, of the routing logic $\mathcal{L}$. The routing function results of the successive application of these unitary moves, that is:

$$\rho_\sharp(s,d) \triangleq \begin{cases} d & \textbf{if } s = d \\ s.\rho_\sharp(\mathcal{L}_\sharp(s,d),d) & \textbf{otherwise} \end{cases} \tag{18}$$

The distance between the current position of a message and its destination is deduced from the topology. The distance between some node s and some node d is noted $dist(s,d)$. Most often, this distance is the measure used to prove that the routing function terminates. It suffices to prove that each unitary move reduces

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

this distance. The distance is a function that returns a natural for any node pair. This function has the following functionality:

$$dist : GenNodeSet_\sharp \times GenNodeSet_\sharp \rightarrow \mathbb{N} \tag{19}$$

To prove the termination of the routing function, $\rho_\sharp$, one has to prove that this function satisfies a concrete instance of proof obligation 2:

$$\forall s, d \in GenNodeSet_\sharp, s \neq d \Rightarrow dist(\mathcal{L}_\sharp(s,d), d) < dist(s,d) \tag{20}$$

The validity of a route is tested by predicate *ValidRoutep*. The definition of *ValidRoutep* is valid for all networks, it needs not be redefined (see Definition 1).

Finally, to validate the concrete routing function, it suffices to prove that is satisfies predicate *ValidRoutep* for the set $NodeSet_\sharp$ of concrete nodes of the network:

$$\begin{aligned}&\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_\sharp)\\ &\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_\sharp(Org_\mathcal{M}(m), Dest_\mathcal{M}(m)), ValidRoutep(r, m, NodeSet_\sharp)\end{aligned} \tag{21}$$

A function that matches the generic definition $Routing_\sharp$ computes a list of routes for each missive of a list $\mathcal{M}$:

Definition 7. Concrete Instance of Function *Routing***.**
$Routing_\sharp(\mathcal{M}, NodeSet_\sharp) \triangleq$

$$\bigwedge_{m \in \mathcal{M}} List(Id_\mathcal{M}(m), Frm_\mathcal{M}(m), \rho_\sharp(Org_\mathcal{M}(m), Dest_\mathcal{M}(m)))$$

To prove the compliance of this function with *GeNoC*, we still need to prove that $Routing_\sharp$ produces a valid travel list if the initial list $\mathcal{M}$ is a valid list of missives:

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_\sharp) \Rightarrow \mathcal{V}_{lstp}(Routing_\sharp(\mathcal{M}, NodeSet_\sharp)) \tag{22}$$

We apply this methodology to the Octagon network presented in Section 2.1.

5.3 Octagon Case Study

Octagon Node Definition Our Octagon model considers an arbitrary, but finite, number of nodes, noted *NumNode*. This number is a natural, multiple of 4. So, we can define that number using a natural N, $NumNode = 4N$. Predicate $ValidNodep_{Oct}$ takes as arguments a node x and number N:

$$\forall N \in \mathbb{N}, \forall x, ValidNodep_{Oct}(x, N) \Leftrightarrow x \in \mathbb{N} \wedge x < 4N \tag{23}$$

Octagon Routing Function Let s be the current node and d the destination node. The three unitary moves in the Octagon are defined as:

$$\mathit{Clockwise}(s, \mathit{NumNode}) \triangleq (s+1) \bmod \mathit{NumNode}$$

$$\mathit{CounterClockwise}(s, \mathit{NumNode}) \triangleq (s-1) \bmod \mathit{NumNode}$$

$$\mathit{Across}(s, \mathit{NumNode}) \triangleq (s + \frac{NumNode}{2}) \bmod \mathit{NumNode}$$

These moves are grouped into function $\mathcal{L}_{Oct}$. The relative address is $\mathit{RelAd} = (d-s) \bmod 4N$. If the current node is the destination, the message is consumed. If the relative address is positive and less than N, the message moves clockwise. If this address is between $3N$ and $4N$, it moves counterclockwise. Otherwise, it moves across. The definition of $\mathcal{L}_{Oct}$ is as follows:

Definition 8. Unitary moves in the Octagon.

$$\mathcal{L}_{Oct}(s, d, N) \triangleq \begin{cases} s & \textbf{if } \mathit{RelAd} = \mathit{0} \\ \mathit{Clockwise}(s, \mathit{4N}) & \textbf{if } 0 < \mathit{RelAd} \leq N \\ \mathit{CounterClockwise}(s, \mathit{4N}) & \textbf{if } 3N \leq \mathit{RelAd} < 4N \\ \mathit{Across}(s, \mathit{4N}) & \textbf{otherwise} \end{cases}$$

Routing function ρ_{Oct} is defined as the recursive application of the unitary moves:

Definition 9. Routing Function of the Octagon, ρ_{Oct}.

$$\rho_{Oct}(s, d, N) \triangleq \begin{cases} d & \textbf{if } s = d \\ s.\rho_{Oct}(\mathcal{L}_{Oct}(s, d, N), d, N) & \textbf{otherwise} \end{cases}$$

As there are two ways of traversing the Octagon, there exist two distances between two nodes. The measure used to prove that function ρ_{Oct} terminates is the minimum between these two distances:

$$\mathit{mes}_{Oct}(s, d, \mathit{NumNode}) = \mathit{Min}[(d-s) \bmod \mathit{NumNode}, (s-d) \bmod \mathit{NumNode}]$$

To prove that the octagon routing function terminate, it suffices to prove that the unitary moves reduce this distance:

Theorem 4. Octagon Routing Function Terminates.
$\forall s, d \in \mathit{GenNodeSet}_{Oct}, s \neq d \Rightarrow$

$$\mathit{mes}_{Oct}(\mathcal{L}_{Oct}(s, d), d, \mathit{NumNode}) < \mathit{mes}_{Oct}(s, d, \mathit{NumNode})$$

Proof. The proof is decomposed according to the different moves. Each one of them reduces the distance. The proof is a huge case split because of functions Min and mod. In ACL2, the proof is decomposed in more that 1200 cases. It only requires 10 additional lemmas about function modulo in addition to the latest arithmetic library [18]. Two lemmas are also required to drive ACL2 to the right case split. The proof is automatically performed in less that 100 seconds on a Pentium IV 1,6 GHz, 256 MB of memory and running under Linux.

To show that function ρ_{Oct} constitutes a valid instance of the generic routing function, we need to prove that it produces routes which satisfy predicate *ValidRoutep*:

Theorem 5. Validity of Octagon Routes.

$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct})$
$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_{Oct}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)),\ ValidRoutep(r, m, NodeSet_{Oct})$

Proof. By induction on the route length.

Finally, function $Routing_{Oct}$ follows the generic signature:

Definition 10. Octagon Routing, function $Routing_{Oct}$
$Routing_{Oct}(\mathcal{M}, NodeSet_{Oct}) \triangleq$

$$\bigwedge_{m \in \mathcal{M}} List(Id_{\mathcal{M}}(m), Frm_{\mathcal{M}}(m), List(\rho_{Oct}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m))))$$

We still need to prove that this function produces a valid travel list. The proof of the following theorem is trivial:

Theorem 6. Type of Octagon Routes.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct}) \Rightarrow \mathcal{V}_{lstp}(Routing_{Oct}(\mathcal{M}, NodeSet_{Oct}))$$

Table 5.3 shows details about the ACL2 modeling and proof. ACL2 is run on a Pentium IV at 1.6 GHz with 256 MB under Linux. The Octagon specification and proof are relatively small, an important point for the initial high level design step. In the proof, a huge amount of time is devoted to arithmetic reasoning.

	Nbr. of functions	Nbr. of theorems	Proof Time (seconds)	Size
OctagonNodeSet	5	4	< 1	70 lines
Lemmas on mod	0	10	< 3	150 lines
Routing	19	41	~ 720	955 lines
Total	21	64	< 740	1325 lines

Table 2. Functions, theorems and proof time for the definition and validation of the Octagon

6 Conclusion and Future Work

We have presented a generic model for communication architectures. It is formalized by function *GeNoC*, which is defined by three key components: interfaces, a routing algorithm and a scheduling policy. The generic model does not assume any particular definition of these components. It only relies on a set of proof obligations (or constraints) associated with each component. The correctness of *GeNoC* includes the proof that messages are either lost or reach their expected destination without modification of their content. This proof is deduced from the proof obligations only. Hence, the specification and the validation of a particular communication architecture amounts to give an *explicit* definition to each component and to prove that these definitions satisfy the corresponding constraints. Moreover, each component is self-contained and can be specified and validated in isolation.

To validate our approach, we have applied it to a variety of architectures that constitute as many concrete instances of our theory: some come from industrial systems, like the AMBA bus or the Octagon network, others are more academic examples, like XY or double Y channel routing in a 2D mesh, packet and circuit switching techniques or the biphase mark protocol Bi-ϕ-M.

The current *GeNoC* definition is very abstract and very simplified. Successive, proven correct refined models are needed before reaching the level of details of an implementation specification. Our work is currently being extended in two different directions. At TIMA, our research involves the application of *GeNoC* to wormhole routing, and the elaboration of a refinement method to derive the correctness of a particular hardware implementation. In Germany, the Verisoft [1] project aims at developing methods and tools for the pervasive verification of computer systems. Theories have already been developed about processors [4], operating systems [9], compilers [16], memories [7], and I/O devices [13]. We aim at verifying a complete distributed system where each node will contain each one of the above components and where nodes are connected through a time triggered bus in a FlexRay flavor. A pencil and paper proof of such a system already exists [3]. From this proof, we have sketched additional constraints on the interfaces to ensure proper communication in a real time context [20]. Ultimately, *GeNoC* will be used as the integration of the different theories in a single framework.

Acknowledgment

The authors would like to thank J Strother Moore, Matt Kaufmann and Warren Hunt for valuable remarks and helpful advices regarding ACL2. We are also thankful to Katell Morin-Allory for suggesting improvements to a previous version of this paper.

References

1. http://www.verisoft.de.
2. H. Amjad. Model Checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September 2004.
3. S. Beyer, P. Böhm, M. Gerke, M. Hillebrand, T. In der Rieden, S. Knapp, D. Leinenbach, and W.J. Paul. Towards the formal verification of lower system layers in automotive systems. In *23nd IEEE International Conference on Computer Design:*

VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings, pages 317–324. IEEE, 2005.
4. S. Beyer, C. Jacobi, D. Kroning, D. Leinenbach, and W.J. Paul. Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 51–65, L'Aquila, Italy, October 2003. Springer-Verlag.
5. R. S. Boyer and J Strother Moore. *A Computation Logic Handbook.* Academic Press, 1988.
6. W. Büttner. Is Formal Verification Bound to Remain a Junior Partner of Simulation? In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, 2005. Invited Speaker.
7. I. Dalinger, M. Hillebrand, and W.J. Paul. On the Verification of Memory Management Mechanisms. In D. Borrione and W.J. Paul, editors, *CHARME 2005*, LNCS. Springer, 2005.
8. W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks.* Morgan-Kaufmann Publisher, 2004.
9. M. Gargano, M. Hillebrand, D. Leinenbach, and W.J. Paul. On the Correctness of Operating System Kernels. In J. Hurd and T. Melham, editors, *TPHOLs 2005*, LNCS. Springer, 2005.
10. B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Rădulescu. Deadlock Prevention in the Æthereal protocol. In D. Borrione and W.J. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 345–348, 2005.
11. K. Goossens, J. Dielissen, and A. Rădulescu. Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, September-October 2005.
12. D. Herzberg and M. Broy. Modeling Layered Distributed Communication Systems. *Formal Aspects of Computing*, 17(1):1–18, 2005.
13. M. Hillebrand, T. In der Rieden, and W.J. Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings*, pages 309–316. IEEE, 2005.
14. K. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking Systems On Chip. *IEEE Micro*, pages 36–45, September-October 2002.
15. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *ACL2 Computer Aided Reasoning: An Approach.* Klulwer Academic Press, 2000.
16. D. Leinenbach, W.J. Paul, and E. Petrova. Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, 2005.
17. J Strother 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.
18. W. A. Hunt R. Krug and J Strother Moore. Linear and Nonlinear Arithemetic in ACL2. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 51–65, L'Aquila, Italy, October 2003. Springer-Verlag.
19. A. Roychoudhury, T. Mitra, and S.R. 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.

20. J. Schmaltz. A Formal Model of Lower System Layer. In *Formal Methods in Computer-Aided Design (FMCAD'06)*, San Jose, CA, USA, November 12-16 2006. IEEE/ACM. (To appear).
21. 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.
22. 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.
23. J. Schmaltz and D. Borrione. A Functional Approach to the Formal Specification of Networks on Chip. In A.J. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 52–66, Austin, Tx, USA, November 2004. Springer-Verlag.
24. 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.
25. J. Schmaltz and D. Borrione. Towards a Formal Theory of On Chip Communications in the ACL2 Logic. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, San Jose, California, USA, August 2006. ACM.
26. G. Spirakis. Beyond Verification: Formal Methods in Design. In A. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, Austin, Texas, USA, November 2004. Springer-Verlag. Invited Speaker.