DOI 10.1007/s00165-007-0049-0
BCS © 2007
Formal Aspects of Computing (2008) 20: 241–258

Formal Aspects of Computing

A functional formalization of on chip communications

Julien Schmaltz[1] and Dominique Borrione[2]

[1]Radboud University, Institute for Computing and Information Sciences, Postbus 9010, 6500 GL Nijmegen, The Netherlands.
E-mail:julien@cs.ru.nl
[2]TIMA Laboratory, VDS Group, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.
E-mail: Dominique.Borrione@imag.fr

Abstract. This paper presents a formal model and a systematic approach to the validation of communication architectures at a high level of abstraction. 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 any modification of their content. The model identifies the key constituents common to *all* on chip communication architectures, and their essential properties from which the correctness theorem is deduced. Each constituent is represented by a function that 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 illustrate our approach on several architectures that constitute concrete instances of the generic $GeNoC$ model. Some of these applications come from industrial designs, such as the AMBA AHB bus or the Octagon network from ST Microelectronics.

Keywords: Networks on chip; Communication architectures; Formal methods; Automated theorem proving

1. Introduction

Current chip technology (65 nm) 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 [Büt05]. 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 [Spi04].

Until recently, most of the verification effort was spent on the processing elements, and the literature specifically devoted to the embedded communication 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 [McM93] to debug an academic implementation of the AMBA AHB protocol [RMK03]. 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 [Amj04] used a model checker, implemented in the HOL [Gor87] 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

Correspondence and offprint requests to: J. Schmaltz, E-mail: julien@cs.ru.nl

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. [GVZ05], who recently specified the Æthereal protocol [GDR05] of Philips in the PVS logic [ORS92]. 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 Intellectual Properties (IPs) and platform based design, the current trend in the SoC design community is to raise the level of abstraction [Spi04] 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 textbooks (e.g. [DaT04]) *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 [Moo93] defined a formal model of asynchrony by a function in the Boyer–Moore logic [BoM88], and showed how to use this general model to verify a biphase mark protocol. More recently, Herzberg and Broy [HeB05] 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.

Our long term objective 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. 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. Our research only deals with these restricted communications systems: buses and NoCs.

Our goal is to 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 the best of our knowledge, a first proposal for a formal theory of communication architectures. We formalize a generic communication architecture in a functional form. The heart of the model is function $GeNoC$, which formalizes the interactions between the three key constituents: interfaces, routing and scheduling. This model is expressed in the computational logic of a theorem proving system. We formally verify the correctness of industrial designs under this model.

The original contributions of our work are threefold: (1) the generic model. It 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. (2) 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 descriptions 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 model. (3) The application of our approach has been demonstrated on industrial designs.

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 instantiation

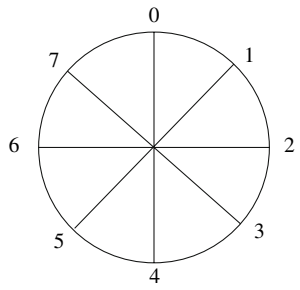

Fig. 1. Basic Octagon Unit

of the $GeNoC$ model to the "Octagon" design by STMicroelectronics is used as an illustration. Finally, Sect. 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 [KND02]. A basic Octagon unit consists of eight nodes and twelve bidirectional links (Fig. 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 [KND02].

An *Octagon packet* is data that must be carried from the source node to the destination node as the 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 Fig. 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. Each node is divided in an application and an interface [RSV97]. Applications represent the computational and functional aspects of nodes. Interfaces represent the communication aspects. To distinguish between interface–application and interface–interface communications, an interface and an application communicate using *messages*; two interfaces communicate using *frames*.

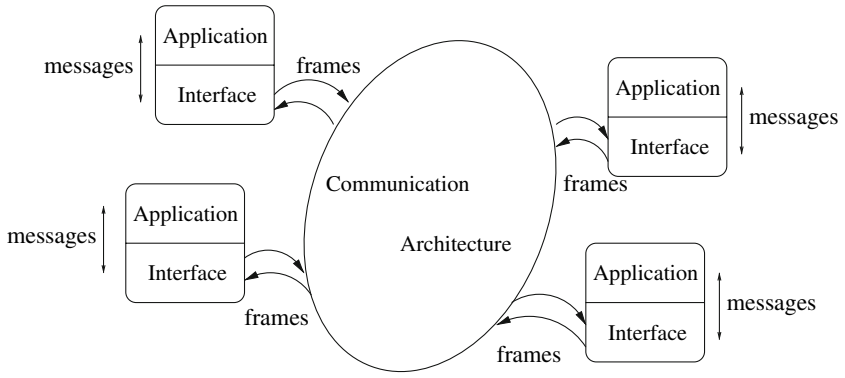

Fig. 2. Communication Model

Table 1. Notations and functions used to manipulate lists

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, $0 \leqslant i \leqslant Len(l) - 1$

Applications 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 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. For instance, 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:

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

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

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

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:

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

Since there is no ambiguity in the membership operator, the same notation will be used when l is a set.

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 simultaneous 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

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

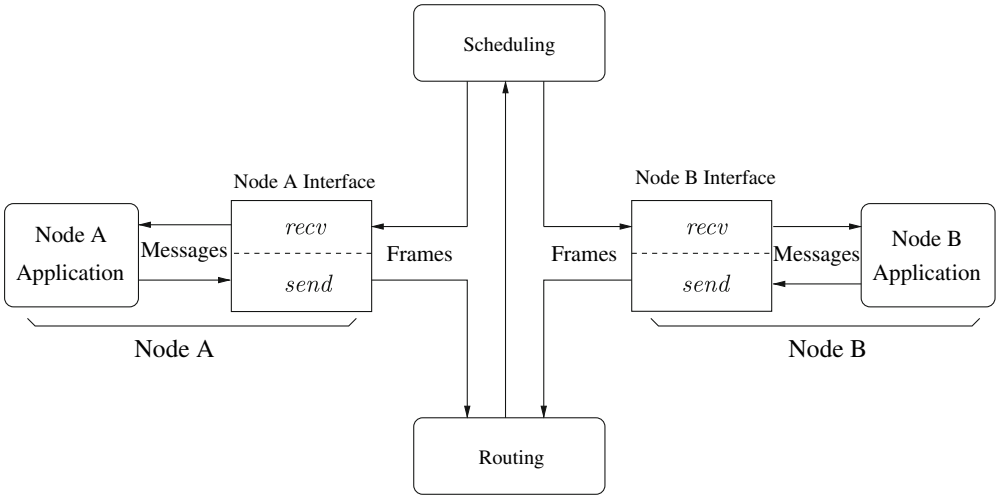

Fig. 3. *GeNoC*: A Generic Network

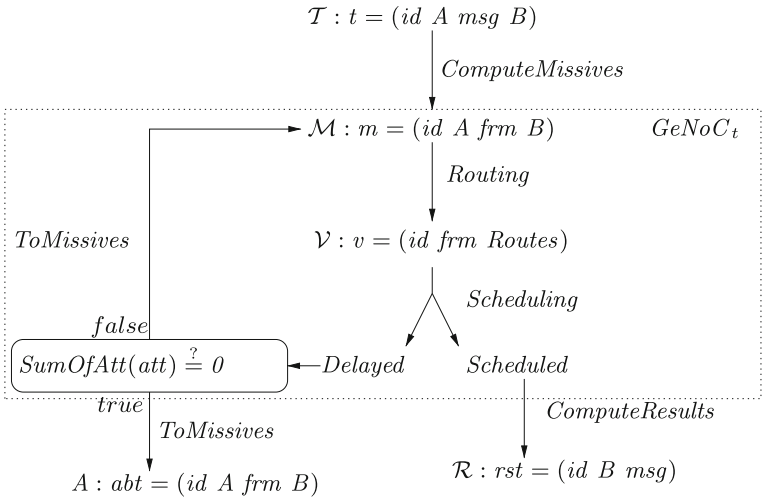

Fig. 4. Unfolding of function *GeNoC*

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 in 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.

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 Fig. 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 $recv$. The applications of functions $Routing$ and $Scheduling$ are combined together. This defines function $GeNoC_t$ (see Sect. 4.5). Delayed travels are converted back to missives and constitute the argument of a recursive call to $GeNoC_t$.

To make sure that this function terminates, we associate a *finite* number of attempts to every node. At every recursive call of $GeNoC_t$, 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 $GeNoC_t$ halts if all attempts have been consumed.

The first output list $\mathcal{R}$ of $GeNoC$ 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 canceled transactions.

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.

$$\forall\, rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \left\{ \begin{array}{ll} & Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge & Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge & Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{array} \right. \tag{4}$$

4. Details of the functional model

4.1. Nodes and parameters

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

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

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:

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

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

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

$$\forall\, msg \in \mathcal{D}_{msg},\ recv \circ send(msg) = msg \tag{PO1}$$

4.3. Routing

4.3.1. 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:

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

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. The principle is that the routing logic is now represented as relation $\mathcal{L}_{ndet}$ which computes a set of possible successors. We obtain all possible routes between two nodes by the recursive application of this relation to every possible successor. This is expressed as follows:

$$\rho_{ndet}(s, d) \triangleq \begin{cases} d & \text{if } s = d \\ \bigwedge_{n \in GenNodeSet, \mathcal{L}_{ndet}(s,n)} n.\rho_{ndet}(\mathcal{L}_{ndet}(n, d), d) & \text{otherwise} \end{cases} \tag{9}$$

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:

$$\rho : GenNodeSet \times GenNodeSet \rightarrow \mathcal{C} \tag{10}$$

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}.

$$\begin{array}{l} \forall\, s, d \in GenNodeSet, \exists\, mes : GenNodeSet \times GenNodeSet \rightarrow S, \\ s \neq d \Rightarrow mes(\mathcal{L}(s, d), d) \prec_S mes(s, d) \end{array} \tag{PO2}$$

When considering adaptive algorithms, the termination condition must hold for all successors proposed by the routing logic.

Proof Obligation 3 Termination Condition for ρ_{ndet}.

$$\begin{array}{l} \forall\, s, d \in GenNodeSet, \exists\, mes : GenNodeSet \times GenNodeSet \rightarrow S, \\ s \neq d \Rightarrow \forall\, n \in GenNodeSet, \mathcal{L}_{ndet}(s, n), mes(n, d) \prec_S mes(s, d) \end{array} \tag{PO3}$$

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.

$$ValidRoutep(r, m, NodeSet) \triangleq \left\{ \begin{array}{ll} & r[0] = Org_{\mathcal{M}}(m) \\ \wedge & Last(r) = Dest_{\mathcal{M}}(m) \\ \wedge & r \subseteq_l NodeSet \wedge Len(r) \geqslant 2 \end{array} \right.$$

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 4 Correctness of routes produced by ρ.

$$\begin{array}{l} \forall\, \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \\ \Rightarrow \forall\, m \in_l \mathcal{M}, \forall\, r \in_l \rho(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)),\ ValidRoutep(r, m, NodeSet) \end{array} \quad \text{(PO4)}$$

4.3.2. 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:

$$Routing : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \rightarrow \mathcal{D}_{\mathcal{V}} \tag{11}$$

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 5 Type of *Routing*.

$$\forall\, \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(Routing(\mathcal{M}, NodeSet)) \quad \text{(PO5)}$$

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.*

$$\begin{array}{l} \forall\, \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow \\ \forall\, v \in_l Routing(\mathcal{M}, NodeSet), \exists! m \in_l \mathcal{M}, Id_{\mathcal{V}}(v) = Id_{\mathcal{M}}(m) \wedge Frm_{\mathcal{V}}(v) = Frm_{\mathcal{M}}(m) \end{array} \quad \text{(TH1)}$$

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.*

$$\forall\, \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow ToMissives \circ Routing(\mathcal{M}, NodeSet) = \mathcal{M} \quad \text{(TH2)}$$

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 returns a new list of number of attempts and two travel lists: the list *Scheduled* and the list *Delayed*. The functionality of *Scheduling* is:

$$Scheduling : \mathcal{D}_{\mathcal{V}} \times AttLst \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{V}} \times AttLst \quad (12)$$

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:

$$\pi_i^j : \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_j \rightarrow \mathcal{D}_i \quad (13)$$

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 6 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* :

$$\forall\, \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \mathcal{V}_{lstp}(Scheduled) \wedge \mathcal{V}_{lstp}(Delayed) \quad \text{(PO6)}$$

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 7 Function *Scheduling* consumes at least one attempt.

Let *natt* be $\pi_3^3 \circ Scheduling(\mathcal{V}, att)$, then:

$$\begin{array}{ll} & SumOfAtt(att) \neq 0 \\ \rightarrow & SumOfAtt(natt) < SumOfAtt(att) \end{array} \quad \text{(PO7)}$$

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 8 Correctness of the delayed travels.

Let *Delayed* be $\pi_2^3 \circ Scheduling(\mathcal{V}, att)$, then:

$$\forall\, \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall\, dtr \in_l Delayed, \exists! v \in_l \mathcal{V}, \left\{ \begin{array}{ll} & Id_{\mathcal{V}}(dtr) = Id_{\mathcal{V}}(v) \\ \wedge & Frm_{\mathcal{V}}(dtr) = Frm_{\mathcal{V}}(v) \\ \wedge & Routes_{\mathcal{V}}(dtr) = Routes_{\mathcal{V}}(v) \end{array} \right. \quad \text{(PO8)}$$

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

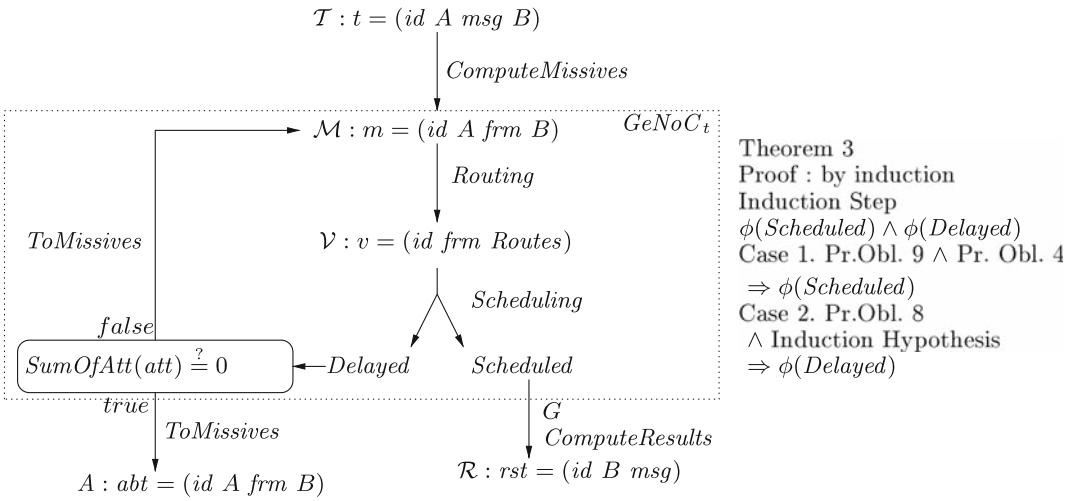

Fig. 5. Proof of $GeNoC$

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

$$\forall\, \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall\, str \in_l Scheduled, \exists! v \in_l \mathcal{V}, \begin{cases} & Id_{\mathcal{V}}(str) = Id_{\mathcal{V}}(v) \\ \wedge & Frm_{\mathcal{V}}(str) = Frm_{\mathcal{V}}(v) \\ \wedge & Routes_{\mathcal{V}}(str) \sqsubseteq Routes_{\mathcal{V}}(v) \end{cases} \quad \text{(PO9)}$$

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 10 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* :

$$\forall\, \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow Delayed_{\lfloor id} \sqcap Scheduled_{\lfloor id} = \epsilon \quad \text{(PO10)}$$

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

$$GeNoC_t : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \times AttLst \times D_{\mathcal{V}} \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{M}} \quad (14)$$

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 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 scheduled 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. Let ϕ be that property. The main idea of the proof of ϕ is pictured in the right part of Fig. 5. The proof proceeds by induction. Since travels in G may have been "delayed" in intermediate recursive calls, in the induction step we need to prove that ϕ is satisfied for the scheduled (case 1) and the delayed travels (case 2). This is expressed by the theorem below:

Theorem 3 *Correctness of* $GeNoC_t$.

$$\forall\ ctr \in_l G, \exists! m \in_l \mathcal{M}, \left\{ \begin{array}{ll} & Id_{\mathcal{V}}(ctr) = Id_{\mathcal{M}}(m) \\ \wedge & Frm_{\mathcal{V}}(ctr) = Frm_{\mathcal{M}}(m) \\ \wedge & \forall\ r \in_l Routes_{\mathcal{V}}(ctr), Last(r) = Dest_{\mathcal{M}}(m) \end{array} \right. \tag{TH3}$$

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 10 that the scheduled and the delayed travels can be proven separately. Scheduled travels have a correspondence 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*. The delayed travels are proven using the induction hypothesis and proof obligation 8. □

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:

$$GeNoC : \mathcal{D}_{\mathcal{T}} \times \mathcal{P}(GenNodeSet) \times AttLst \rightarrow \mathcal{D}_{\mathcal{R}} \times \mathcal{D}_{\mathcal{M}} \tag{15}$$

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:

$$ComputeMissives : \mathcal{D}_{\mathcal{T}} \rightarrow \mathcal{D}_{\mathcal{M}} \tag{16}$$

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:

$$ComputeResults : \mathcal{D}_{\mathcal{V}} \rightarrow \mathcal{D}_{\mathcal{R}} \tag{17}$$

It is defined as follows:

[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.

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

$$\forall\, rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \begin{cases} & Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge & Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge & Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{cases} \quad \text{(TH4)}$$

Proof. The last term of the conjunct is directly obtained from Lemma 3. 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 [KMM00]. 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.
- The ACL2 community continuously extends and improves the system. 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 [ScB06] for details), the complete $GeNoC$ formalization could be performed in the ACL2 logic, using the above listed capabilities, 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 1,864 lines of code. Only one fourth of these is dedicated to the encapsulation of the different modules. Most of the definitions

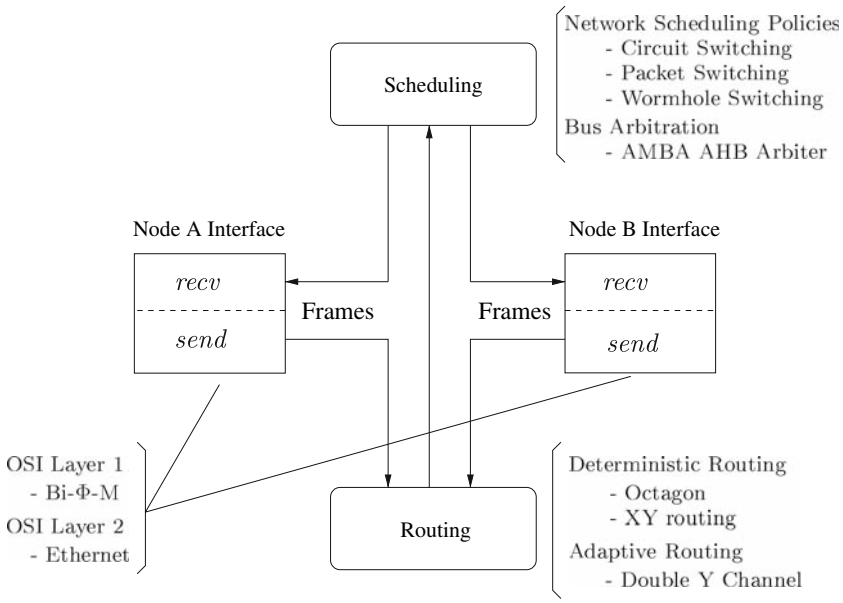

Fig. 6. Concrete Instances of $GeNoC$

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

Figure 6 summarizes concrete instances of on chip communications constituents in our model. Any combination of these different concrete instances is defined and validated by generic function $GeNoC$, which means without any additional effort.

We have shown that the circuit [ScB04] and the packet [ScB05] switching techniques are concrete instances of $Scheduling$. Recently, we have modeled a wormhole based network [BHP07]. Based on previous work [ScB03], 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 [Moo93], we proved that his model of the biphase protocol constitutes a valid instance of the interfaces. We have modeled an Ethernet controller[3] and we are investigating its compliance with $GeNoC$.

With respect to $Routing$, several topologies and associated routing algorithms were shown to be correct instances of this function. A model and a proof of the Octagon [ScB04], as well as the XY routing in a 2D mesh [ScB05] are such valid instances of our generic model. 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 [Sch06].

Among all these industrial cases, we choose to illustrate our methodology on the Octagon network. The presentation that follows, based on the generic model, is significantly different from the earlier proof [ScB04].

5.2. Method for instantiating the model

The application of $GeNoC$ to a particular network consists of the following steps:

Node Definition. The topology of a network determines the node numbering and the unitary moves allowed between two adjacent nodes. Before all, we define the node definition domain, which is a particular instance

[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.

of predicate $ValidNodep$, noted $ValidNodep_\sharp$. The generic definition domain $GenNodeSet$ becomes a particular domain $GenNodeSet_\sharp$, the naturals for instance. We give a concrete definition of Eq. 5, that is:

$$\forall\, x,\ ValidNodep_\sharp(x) \Leftrightarrow x \in GenNodeSet_\sharp \tag{18}$$

Routing Definition. First, we identify the moves allowed between two adjacent nodes. As we consider regular networks (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 $\rho_\sharp$ results of the successive application of these unitary moves (see Eq. 8, or Eq. 9 in the adaptive case).

The distance between the current position of a message and its destination is deduced from the topology. The distance between nodes s and 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 this distance. The distance is a function that returns a natural for any node pair. This function has the following functionality:

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

To prove the termination of $\rho_\sharp$, we must prove that this function satisfies a concrete instance of proof obligation PO2 (or PO3 in the adaptive case).

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, instantiating proof obligation PO4.

A function $Routing_\sharp$ that matches the generic definition 2 (of Sect. 4.3) computes a list of routes for each missive of a list $\mathcal{M}$.

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 (proof obligation PO5).

Please recall that all these proof obligations are automatically generated by ACL2 from the instantiation of the generic functions.

Scheduling. The instantiation of the generic $Scheduling$ function follows a similar pattern. A concrete definition must be provided, and proof obligations PO6 to PO10 must be discharged.

Main Function. The definition and proof of $GeNoC_{t_\sharp}$ and $GeNoC_\sharp$ follow from the above. They can be automatically macro-generated in ACL2.

5.3. Octagon case study

5.3.1. 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:

$$\forall\, N \in \mathbb{N}, \forall\, x,\ ValidNodep_{Oct}(x, N) \Leftrightarrow x \in \mathbb{N} \wedge x < 4N \tag{20}$$

5.3.2. Octagon routing function

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

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

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

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

These moves are grouped into function $\mathcal{L}_{Oct}$. The relative address is $RelAd = (d - s) \bmod 4N$. 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 7 Unitary moves in the Octagon.

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

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

Definition 8 Routing Function of the Octagon, ρ_{Oct}.

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

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:

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

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

Theorem 5 *Octagon Routing Function Terminates.*

$$\forall\, s, d \in GenNodeSet_{Oct}, s \neq d \Rightarrow mes_{Oct}(\mathcal{L}_{Oct}(s, d), d, NumNode) < mes_{Oct}(s, d, NumNode) \tag{TH5}$$

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 1,200 cases. It only requires 10 additional lemmas about function modulo in addition to the latest arithmetic library [RKM03]. Two lemmas are also required to drive ACL2 to the right case split. The proof is automatically performed in less that 100 s 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 6 *Validity of Octagon Routes.*

$$\begin{aligned} &\forall\, \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct}) \\ &\quad \Rightarrow \forall\, m \in_l \mathcal{M}, \forall\, r \in_l \rho_{Oct}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet_{Oct}) \end{aligned} \tag{TH6}$$

Proof. By induction on the route length. □

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

Definition 9 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 7 *Type of Octagon Routes.*

$$\forall\, \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct}) \Rightarrow \mathcal{V}_{lstp}(Routing_{Oct}(\mathcal{M}, NodeSet_{Oct})) \tag{TH7}$$

Table 2 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.

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

	No. of functions	No. of theorems	Proof time (s)	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	1,325 lines

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 an *explicit* definition of each component and the proof 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, wormhole 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. We are currently investigating different extensions of $GeNoC$. Our research involves the further application of $GeNoC$ to NoC designs, especially designs that can be synthesized on FPGA platforms. We are interested in extending $GeNoC$ with control flow mechanisms, like credits [DaT04].

Our current treatment of adaptive routing algorithms does not take into account the current global state of the network. The present routing function computes *all* possible routes. If this works for minimal algorithms, it might not be feasible for non-minimal ones. We will surely face a similar issue when investigating fault-tolerance. More abstract previous works targetted at protocols (e.g. [MGP04]) do not consider the underlying interconnect structure: in that respect our work would bring complementary results.

We plan to extend $GeNoC$ with a global notion of the network state. Finally, we are studying the extension of $GeNoC$ to consider low level aspects, like metastability or clock drift and jitter. The first author has recently defined a formal model of low level system layers [Sch07], which considers these phenomena: from this work, additional proof obligations will be identified, to guarantee the correctness of communications while considering these aspects.

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. We would like to thank our anonymous reviewers for providing apposite remarks to enhance the presentation of this paper.

References

[Amj04] Amjad H (2004) Model checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September, UK

[Büt05] Büttner W (2005) Is formal verification bound to remain a junior partner of simulation? In: Borrione D, Paul W (eds) Correct hardware design and verification methods (CHARME'05), Volume 3725 of LNCS Invited Speaker. Springer, Saarbrücken

[BHP07] Borrione D, Helmy A, Pierre L, Schmaltz J (2007) A generic model for formally verifying NoC communication architectures: a case study. In: Proceedings of first international symposium on networks-on-chip (NOCS'07), IEEE, Princeton, 7–9 May, pp 127–136

[BoM88] Boyer RS, Strother MJ (1988) A computation logic handbook. Academic, New York
[DaT04] Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan-Kaufmann, San Fransisco
[GDR05] Goossens K, Dielissen J, Rădulescu A (2005) Æthereal network on chip: concepts, architectures, and implementations. IEEE Des Test Comput 22(5):414–421
[Gor87] Gordon MJC (1987) HOL: a proof generating system for higher-order logic. In: Birthwislte G, Subrahmanyam PA (eds) VLSI specification, verification and synthesis. Kluwer, Boston, pp 73–128
[GVZ05] Gebremichael B, Vaandrager F, Zhang M, Goossens K, Rijpkema E, Rădulescu A (2005) Deadlock Prevention in the Æthereal protocol. In: Borrione D, Paul WJ (eds) Correct hardware design and verification methods (CHARME'05), Volume 3725 of LNCS. Springer, Heidelberg, pp 345–348
[HeB05] Herzberg D, Broy M (2005) Modeling layered distributed communication systems. Form Asp Comput 17(1):1–18
[KMM00] Kaufmann M, Manolios P, Strother Moore J (2000) ACL2 computer aided reasoning: an approach. Klulwer, Dordrecht
[KND02] Karim K, Nguyen A, Dey S (2002) An interconnect architecture for networking systems on chip. IEEE Micro, September–October, pp 36–45
[McM93] McMillan KL (1993) Symbolic model checking. Kluwer, Dordrecht
[MGP04] Miner PS, Geser A, Pike L, Maddalon J (2004) A unified fault-tolerance protocol. In: Lakhnech Y, Yovine S (eds) Formal techniques, modeling and analysis of timed and fault-tolerant systems (FORMATS-FTRTFT), Volume 3253 of LNCS. Springer, Heidelberg, pp 167–182
[Moo93] Strother Moore J (1994) A formal model of asynchronous communications and its use in mechanically verifying a biphase Mark Protocol. Form Asp Comput 6(1):60–91
[ORS92] Owre S, Rushby JM, Shankar N (1992) PVS: a prototype verification system. In: Kapur D (ed) Eleventh international conference on automated deduction (CADE'92), Saragota, Volume 607 of LNAI. Springer, Heidelberg, pp 748–752
[RKM03] Hunt WA, Krug R, Strother Moore J (2003) Linear and nonlinear arithemetic in ACL2. In: Geist D, Tronci E (eds) Correct hardware design and verification methods (CHARME'03), Volume 2860 of LNCS. Springer, L'Aquila, pp 51–65
[RMK03] Roychoudhury A, Mitra T, Karri SR (2003) Using formal techniques to debug the AMBA system-on-chip Bus Protocol. In: Design automation and test Europe (DATE'03), pp 828–833
[RSV97] Rowson JA, Sangiovanni-Vincentelli A (1987) Interface-based design. In: 34th design automation conference (DAC'96), pp 178–183
[ScB03] Schmaltz J, Borrione D (2003) Verification of a parameterized bus architecture using ACL2. In: Proceedings of the fourth international workshop on the ACL2 theorem prover and its applications
[ScB04] Schmaltz J, Borrione D (2004) A functional approach to the formal specification of networks on chip. In: Hu AJ, Martin AK (eds) Formal methods in computer-aided design (FMCAD'04), Volume 3312 of LNCS. Springer, Austin, pp 52–66
[ScB05] Schmaltz J, Borrione D (2005) A generic network on Chip Model. In: Melham T, Hurd J (eds) Theorem proving in higher order logics (TPHOLs'05), Volume 3603 of LNCS. Springer, Oxford, pp 310–325
[ScB06] Schmaltz J, Borrione D (2006) 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, part of FloC'06. ACM, Seattle, pp 47–60
[Sch06] Schmaltz J (2006) Une formalisation fonctionnelle des communications sur la puce. PhD thesis, Joseph Fourier University, Grenoble, France (in French). A partial translation is available upon request to the first author
[Sch07] Schmaltz J (2007) A formal model of clock domain crossing and automated verification of time-triggered hardware. In: Baumgartner J, Sheeran M (eds) Formal methods in computer-aided design (FMCAD'07). IEEE/ACM, Austin (to appear)
[Spi04] Spirakis G (2004) Beyond verification: formal methods in design. In: Hu A, Martin AK (eds) Formal methods in computer-aided design (FMCAD'04), Volume 3312 of LNCS. Springer, Austin, USA Invited Speaker

Received 12 September 2006
Revised 23 June 2007
Accepted 10 September 2007 by C. Delgado Kloos
Published online 18 October 2007