Executable Formal Specification and Validation of NoC Communication Infrastructures

Dominique Borrione, Amr Helmy, Laurence Pierre
TIMA Laboratory (CNRS-GrenobleINP-UJF)
46 Av. Félix Viallet
Grenoble, France
name.surname@imag.fr

Julien Schmaltz
Radboud University Nijmegen
Institute for Computing and Information Sciences
P.O. Box 9010
Nijmegen, The Netherlands
julien@cs.ru.nl

ABSTRACT
We describe an enhanced generic model for Networks-on-Chip (NoCs), implemented in the executable logic of the ACL2 theorem prover. The model is meant for serving as a formal reference for the specification, validation, and simulation at the initial design phase. Instantiated on a specific NoC, the model may be used for formal proofs and for simulation. The methodology is illustrated on HERMES.

Categories and Subject Descriptors
B.7.2 [Integrated Circuits]: Design Aids—simulation, verification

General Terms
Hardware verification

Keywords
Verification, Theorem proving, Simulation

1. INTRODUCTION
The Network-on-Chip (NoC) paradigm is emerging as a promising solution for the design of complex Systems-on-Chips (SoC). The most widespread techniques to analyze the behavior of NoCs are based on simulation or emulation [4, 10]. Formal verification methodologies provide a reliable alternative: mathematical techniques, instead of the dynamic execution of test cases, are used to prove desired properties.

The verification of NoCs needs to be parameterized. Not only the network structure may be designed as a scalable regular repetition of nodes, but also because any node can request a communication of an arbitrary length at any time. In this context, the direct application of automatic techniques (e.g., model checking) is very difficult; deduction-oriented – but interactive – reasoning techniques are needed.

2. GENERIC VERIFICATION METHOD
The approach specifies the transmission of messages on a generic communication architecture, with an arbitrary network characterization (topology and node interfaces), routing algorithm and switching technique.

Proof principles. The modeling and proof principles are illustrated in Fig. 1. The main function of the model, named GeNoC, is constructed from the assembly of key components: topology of the network $T$, interfaces $I$, routing $R$, and scheduling $S$. A correctness theorem is associated with
this function. It states that for all $T$, $I$, $R$, and $S$ that satisfy some given constraints – say $P_1$, $P_2$, $P_3$, and $P_4$ – GeNoC fulfills a correctness property $\varphi$:

\[
\forall T \forall I \forall R \forall S, \quad P_1(T) \land P_2(I) \land P_3(R) \land P_4(S) \Rightarrow \varphi(\text{GeNoC}(T, I, R, S))
\] (1)

Roughly speaking, the property $\varphi$ asserts that every message arrived at a node actually corresponds to a message issued at another node of the network, and the message reaches the intended destination without modification of its content.

The constraints express essential properties of the key components, e.g. well-formedness of the network or termination of the routing function.

In this formula, $T$, $I$, $R$, and $S$ are universally quantified. It follows that, for any instance of a network – i.e., for any $T_0$, $I_0$, $R_0$, and $S_0$ – the property $\varphi(\text{GeNoC}(T_0, I_0, R_0, S_0))$ holds provided that $P_1(T_0)$, $P_2(I_0)$, $P_3(R_0)$, and $P_4(S_0)$ are satisfied. Hence, verifying statement (1) above for a given NoC is reduced to discharging these instantiated constraints (or proof obligations) on the NoC constituents.

Communication abstraction. An arbitrary, but finite, number of nodes are connected to some interconnection structure. Each node is capable of sending and receiving messages, and is made of an application and an interface. Interfaces allow applications to communicate using protocols. Active applications (processors) are reduced to the list of their pending communication operations. Since communications that are local to a node need not use the network, we shall consider that, in every communication, the destination and the source nodes differ.

### 3. THE GENOC MODEL

The model consists in function $\text{GeNoC}$ (Fig. 2). It takes as main argument a list of messages emitted at source nodes and returns a list of messages received at destination nodes. Function $\text{GeNoC}$ entails: a global network state representing the current status of the ports of the nodes, and an explicit notion of time (Section 3.2); interfaces that are represented by two functions used to encode $(p2psend)$ and to decode $(p2prcv)$ messages; the routing algorithm and topology that are represented by function $\text{Routing}$ (Section 3.3); and the switching technique that is represented by function $\text{Scheduling}$ (Section 3.4). These functions are not given explicit definitions. They are characterized by the set of properties they should satisfy, called proof obligations or constraints.

### 3.1 Inputs and outputs of GeNoC

The model considers as its main input, a list of messages to be sent in the network. These messages are first encoded and then injected in the network.

In addition to the data that it is carrying, a message contains the following information: its source, destination, current position, the simulation cycle at which it is injected, and optionally the number of flits in the message. Each message is also uniquely identified by a natural number.

Function $\text{GeNoC}$ produces two output lists: the list of messages that have reached their destination, and the list of messages that are still at their source, or traveling in the network.

### 3.2 Global network state and time

The global state of the network is characterized by the status of the ports of the nodes. We define accessors and update functions necessary to manipulate the state. Most of them are generic.

Our time unit corresponds to one recursive call in function $\text{GeNoC}$. In practice, this represents one computation step, where messages proceed by at most one hop. Different messages can depart at different times. Function $\text{R4D}$ ("ready for departure") determines which messages can be in the network at the current time. Messages may be injected only at specific execution time, or under constraints on the network load (e.g., credit-based flow control). Function $\text{R4D}$ takes as arguments a list $M$ of messages and the current time $z$. It returns two lists: a list $\text{TR}$ of traveling messages allowed in the network, the rest of the messages constitute the delayed messages $\text{D}$.
3.3 Routing algorithm

The routing algorithm is represented by the successive application of unitary moves (e.g., routing hops). For each message, the routing function computes all possible routes from the current position to the destination.

3.4 Switching technique

Function Scheduling returns a list of messages that have reached their destination, a list of messages that are en route to their destination, and a new state. The decision of moving one message from one port to another depends on the current state of the network, which itself depends on the other messages. This enables the visualization of deadlocks, and the modeling of lower layer protocols.

Table 1 summarizes the main proof obligations that characterize all these key constituents. Other constraints express for instance simple typing properties.

<table>
<thead>
<tr>
<th>NoC constituent</th>
<th>Main proof obligations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>The composition of the encoding and decoding functions is the identity.</td>
</tr>
<tr>
<td>Network state</td>
<td>Every state modification functions return a valid state. The intersection of lists TR and D produced by R4D is empty.</td>
</tr>
<tr>
<td>Routing</td>
<td>Function Routing terminates (recursion is well-founded). Each route from a current node n to a destination d actually starts in n, uses only valid nodes, and ends in d.</td>
</tr>
<tr>
<td>Scheduling</td>
<td>Arrived and en route messages are well-formed. The intersection of these two lists of messages is empty.</td>
</tr>
</tbody>
</table>

Table 1: Main constraints of the NoC constituents

3.5 Function GeNoC

Function GeNoC makes use of functions R4D, Routing and Scheduling to compute the arrived and the delayed messages. It takes as parameters: a list of messages (M), the structure of the network, reduced to the set of its nodes (NS), a finite number of attempts (att), the set of arrived messages (T, originally empty), the current state of the network (S), and the current time (z). The number of attempts is decremented by 1 at each node with a message waiting for injection. Function SumOfAtt(att) computes the sum of the remaining attempts of the nodes and is used as the decreasing measure of parameter att.

If no attempt is left, GeNoC stops and returns a pair composed of the arrived (T), and the delayed or en route (M) messages. Otherwise, every recursive call processes a list of messages, where some are waiting at their source, and some are traveling in the network. For each traveling message produced by function R4D, function Scheduling computes the list of the arrived messages (A), the list of messages that are still traveling in the network (TM), the remaining attempts (att’), and a new state (S’). The recursive call processes the traveling messages together with the messages (D) delayed by R4D. Time is incremented by 1.

```
GeNoC(M, NS, att, T, S, z) =
if SumOfAttempts(att)=0
then list(T, M)
else
let <TR, D> = R4D(M, z)
in
let <TM, A, att', S'> =
Scheduling(Routing(TR, NS), att, S)
in GeNoC(union(TM, D), NS, att',
union(A, T), S', z+1)
```

Function GeNoC is considered correct if every message
arrived at some node \( n \) was emitted at a valid node of the network, was addressed to node \( n \), and its content equals the content at injection time (property \( \varphi \) of Section 2).

This general proof is done in ACL2 once and for all. Then, proving that a particular NoC satisfies this correctness property amounts to (1) the definition of the functions that describe the network topology, (2) the definition of all the functions associated with the communications, in particular Routing and Scheduling, and (3) the ACL2 proof of the instances of the generic proof obligations for these functions.

4. APPLICATION TO HERMES

HERMES [9] is based on a regular 2D mesh architecture (Fig. 3), and the XY routing algorithm [1] (messages are routed along the X-axis until they reach the first coordinate of their destination, then they travel along the Y-axis). Each switch has routing logic and five bi-directional ports (Fig. 3): East, West, North, South, and Local. Each port has one input buffer. HERMES uses the wormhole switching technique: messages are decomposed into smaller units called flits. The header flit contains information needed for routing. The control flit determines the number of data flit (data payload) that follow in a pipelined fashion.

![Figure 3: Mesh architecture and HERMES switch](image)

4.1 Formal proof

A node is defined as a 4-tuple \((X, Y, P, D)\), where \( X \) and \( Y \) are the coordinates of the node, \( P \in \{E, W, S, N, L\} \) is the port name, and \( D \in \{\text{input, output}\} \) is the direction. \( R4DHermes \) splits the whole list of messages \( M \) into two sub-lists, the delayed messages \( D \) (their departure time is greater than the current time) and the traveling ones \( TR \).

Ten proof obligations and 13 intermediate lemmas have been proven for \( R4D \).

**Routing function.** The Routing function computes the list of possible routes between each current node and destination node for the traveling messages returned by function \( R4DHermes \). Its input parameter is a list of messages, and it returns a list of routed messages. Here is our formalization of the XY routing algorithm in HERMES:

\[
\text{XYRouting}(M) = \\
\text{if empty}(M) \text{ then nil} \\
\text{else let mess}=\text{first}(M) /*\text{ first message */} \\
\text{and from}=\text{Org}(mess) /*\text{ origin */} \\
\text{and cur}=\text{Cur}(mess) /*\text{ current node */} \\
\text{and to}=\text{Dest}(mess) /*\text{ destination */} \\
\text{and id}=\text{Id}(mess) /*\text{ identifier */} \\
\text{and data}=\text{Data}(mess) /*\text{ content */} \\
\text{and flits}=\text{Flit}(mess) /*\text{ # flits */} \\
\text{let tr}=\text{first}(L) /*\text{ first message */} \\
\text{and r}=\text{routesOf}(tr) /*\text{ set of routes */} \\
\text{and c}=\text{check_routes}(r) /*\text{ valid route */} \\
\text{and a}=\text{check_arrival}(c) /*\text{ dest. reached */} \\
\text{in if c!=nil /* the message can move */} \\
\text{then let TM'}=\text{if a then} \text{TM else} \text{union(update(tr),TM)} \\
\text{and A'}=(\text{if a then union(tr,A) else A) in} \\
\text{WormHSched(rest(L),TM',A') } \\
\text{else let TM'}=\text{if a then} \text{TM else} \text{union(update(tr),TM)} \\
\text{and A'}=(\text{if a then union(tr,A) else A) in} \\
\text{WormHSched(rest(L),TM',A',updateSt(S))} \\
\text{else WormHSched(rest(L),union(tr,TM'),A,S)} \\
\text{Figure check_routes checks if there exists a route compatible with the current state of the network. If this check succeeds and the message has arrived to its destination, it is added to list A, and the state of the network is updated. If the message is still en route or if there is no available route, the message is added to TM and the state is left unchanged. Fifteen proof obligations have been proven for WormHoleSched, using 15 intermediate lemmas.}

**Instantiation of GeNoC.** Function \( \text{GeNoC} \) instantiated for the HERMES network corresponds to the generic one given in Section 3, where \( R4D \), Routing and Scheduling are instantiated with the corresponding HERMES functions:

\[
\text{GeNoC}(M, NS, att, T, S, z) = \\
\text{if SumOfAttempts(att)=0 then list(T, M) else} \\
\text{let <TR, D> } = \text{R4DHermes(M,z) in} \\
\text{let <TM, A, att', S'} = \text{WormHoleSched(XYRouting(TR,NS),att,S) in GeNoC(union(TM,D),NS,att',union(A,T),S',z+1)}/
\]
We obtain the correctness theorem for HERMES by instantiating statement (1) of Section 2. This proof is generic on the size of the 2D-mesh, on the number of messages, on their numbers of flits and on their injection times.

4.2 Simulation

ACL2 provides both a theorem prover and an execution engine in the same environment. Comparative VHDL/ACL2 simulations with the initial GeNoC model were reported in [2]. Due to the limitations of this earliest model, message blocking could not be formalized faithfully in ACL2: in the VHDL and ACL2 simulations, messages went through the same nodes and reached the right destination, but were not blocked at the same place. As a result, the state of each message separately was the same in both models, but the global state was different.

The new formalization is more faithful to the VHDL description. Consider the simulation scenario described in Fig. 4. Three messages are present, as shown in Table 2. Message 1 is split into 6 flits (hence its total number of flits is 8), message 2 is split into 3 flits, and message 3 is split into 5 flits.

![Figure 4: Example in a 4x4 mesh](image)

Table 2: Messages of the simulation example

<table>
<thead>
<tr>
<th>ID</th>
<th>Source</th>
<th>Destination</th>
<th>Contents</th>
<th>Nb of flits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2 3)</td>
<td>(3 0)</td>
<td>11 24 31 41 51 61</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>(0 2)</td>
<td>(3 1)</td>
<td>12 22 32</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>(1 2)</td>
<td>(2 1)</td>
<td>13 23 33 43 53</td>
<td>7</td>
</tr>
</tbody>
</table>

VHDL Simulation. The results of the simulation of the VHDL description of HERMES are displayed in Fig. 5. Each node is represented by a signal data_showXY, where X and Y are the node coordinates. Each signal is made of five 8-bit vectors, one for each port: 0 for East, 1 for West, 2 for North, 3 for South, and 4 for Local.

The simulation trace displays the evolution of the three messages. For instance, consider message 3, that originates at node (1 2). Its successive flits contain 21 (destination), 7 (number of flits), and 13, 23, 33, 43, 53 (data payload). Message 3 goes successively through data_show12(0), the East port of node (1 2), data_show22(2), the North port of node (2 2), and data_show21(4), the Local port of node (2 1). Similarly, message 1 goes successively through data_show23(0), the East port of node (2 3), data_show33(2), the North port of node (3 3), data_show32(2), the North port of node (3 2), data_show31(2), the North port of node (3 1), and finally data_show30(4), the Local port of node (3 0). As for message 2, its path must be: data_show02(0), data_show12(0), data_show22(0), data_show32(2), and data_show31(4). Since the East port of node (1 2) is already occupied by message 3 when message 2 arrives, this message is blocked in this node as long as message 3 gets through. Then message 2 completes its travel.

GeNoC Simulation. The contents of the rows of Table 2 correspond to the successive states of the nodes along the recursive calls to GeNoC. Flits are numbered in decreasing order, from nbfllits-1 down to 0. These simulation results are faithful to the ones of the VHDL simulation (Fig. 5). The three messages, which depart simultaneously, proceed in the network concurrently, up to the moment when message 2 is blocked at node (1 2) that is already occupied by message 3 (4th step). Message 2 completes its travel to its destination as soon as this node is freed (10th step).

<table>
<thead>
<tr>
<th>Step</th>
<th>Evolution of the messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(((3 6) (1 2 L I)) (2 4) (0 2 L I)) (1 7) (2 3 L I))</td>
</tr>
<tr>
<td>2</td>
<td>(((3 6) (1 2 E O)) (2 4) (0 2 E O)) (1 7) (2 3 E O))</td>
</tr>
<tr>
<td>3</td>
<td>(((3 6) (2 2 W I)) (2 4) (1 2 W I)) (1 7) (3 3 W I))</td>
</tr>
<tr>
<td>4</td>
<td>(((3 6) (2 2 N O)) (2 4) (1 2 W I)) (1 7) (3 3 N O))</td>
</tr>
<tr>
<td>5</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>6</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>7</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>8</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>9</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>10</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>11</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>12</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>13</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>14</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>15</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>16</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>17</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>18</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>19</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>20</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
<tr>
<td>21</td>
<td>(((3 6) (2 2 S I)) (2 4) (1 2 W I)) (1 7) (3 3 S I))</td>
</tr>
</tbody>
</table>

Table 3: ACL2 simulation

In the first step, flit number 6 of message 3 takes port L of node (1 2), flit number 4 of message 2 takes port L of node (0 2), and flit number 7 of message 1 takes port L of node (2 3). In the second step, two flits of each message are visible: flit 6 of message 3 takes port L of node (1 2) and its flit 5 takes port L of node (2 3) while their flits 3 and 6 follow through ports L of the same nodes. From the 4th step, two flits of each message are visible: flit 6 of message 3 takes port L of node (1 2) and its flit 5 takes port L of node (2 3) while their flits 3 and 6 follow through ports L of the same nodes. From the 10th step, message 2 can move again. After 21 steps, the last flit (number 0) of message 2 reaches the local port of node (3 1).

The ACL2 execution of GeNoC yields such textual results, made of lists. For readability, we have developed an animated visualization tool in Java: from ACL2 results in list form, it performs an animation of the evolution of the messages in the network [2].
5. RELATED WORK

In the context of on-chip communications, several specific architecture have been studied. Roychoudhury et al. use the SMV model checker to debug an academic implementation of the AMBA AHB protocol [11]. They detect a live lock scenario that was caused by the implementation of their arbiter rather than by the protocol itself. Theorem provers, or combinations of theorem provers and model-checkers, have also been exploited. Amjad [1] uses a model checker, implemented in the HOL theorem prover, to verify the AMBA APB and AHB protocols, and their composition in a single system. In Gebremichael et al. [3], the Æthereal protocol of Philips has been specified in the PVS logic. The main property that has been verified is the absence of deadlock for an arbitrary number of masters and slaves.

The research results that we now review tackle the formalization from a generic perspective, and were more influential to our own research. Moore [8] defines a formal model of asynchrony by a function in the Boyer-Moore logic, and shows how to use this general model to verify a biphase mark protocol. More recently, Herzberg and Broy [5] presents 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 define 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.

6. CONCLUSION

The GeNoC model presented for the first time in this paper constitutes a “hop-level” model of the network, where the observation time unit is the elementary move of messages from one node to a neighbor node, according to the network topology. The state of the NoC is explicit, and all issued messages that may move, according to the routing policy, actually perform one hop.

Many more refinement steps are needed before a cycle accurate modeling of the RTL design can be reached. At each step, a proof of compliance between the more detailed model and the more abstract one is needed. Our current efforts aim at providing a systematic method to help the designer and the proof engineer show that design decisions along the refinement path preserve the correct behavior of the network.

7. REFERENCES