Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 548324, 14 pages
doi:10.1155/2009/548324

Research Article

A Formal Approach to the Verification of Networks on Chip

Dominique Borrione,[1] Amr Helmy,[1] Laurence Pierre,[1] and Julien Schmaltz[2]

[1] *Techniques of Informatics and Microelectronics for Integrated Systems Architecture (TIMA) Laboratory (CNRS, GrenobleINP, UJF), 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France*
[2] *Institute for Computing and Information Sciences (ICIS), Radboud University, Postbus 9010, 6500 GL Nijmegen, The Netherlands*

Correspondence should be addressed to Laurence Pierre, laurence.pierre@imag.fr

Received 1 August 2008; Accepted 4 February 2009

Recommended by Gregor Goessler

The current technology allows the integration on a single die of complex systems-on-chip (SoCs) that are composed of manufactured blocks (IPs), interconnected through specialized networks on chip (NoCs). IPs have usually been validated by diverse techniques (simulation, test, formal verification) and the key problem remains the validation of the communication infrastructure. This paper addresses the formal verification of NoCs by means of a mechanized proof tool, the ACL2 theorem prover. A metamodel for NoCs has been developed and implemented in ACL2. This metamodel satisfies a generic correctness statement. Its verification for a particular NoC instance is reduced to discharging a set of proof obligations for each one of the NoC constituents. The methodology is demonstrated on a realistic and state-of-the-art design, the Spidergon network from STMicroelectronics.

Copyright © 2009 Dominique Borrione et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Up-to-date systems-on-chip (SoCs) are built by assembling modules such as processor cores, memories, and specialized Intellectual Property (IP) blocks. Networks on chip (NoCs) are emerging as a promising solution for the realization of communications in such complex systems [1, 2]. NoCs such as Faust [3] or SoCBUS [4] provide configurable communication infrastructures that are particularly attractive for the design of advanced SoCs and embedded systems. NoCs are usually made of point-to-point data links interconnected by switches; widespread topologies include meshes and fat-tree architectures.

SoC design must be supported by appropriate design methods and flows. We place ourselves in the context of "platform-based design," where a platform is defined as "an abstraction layer in the design flow that facilitates a number of possible refinements into a subsequent abstraction layer (platform)" [5]. Verification is indispensable at all stages and layers. It consumes almost 60 to 70% of the total resources in a typical chip design cycle [6]. In an NoC-based architecture, two main verification problems arise: the functional validation of each core or IP and the verification of the communications in the network. This paper addresses the latter aspect.

The application of formal methods to embedded communication infrastructures has received a recent attention. The sparse literature on that topic is targeted to specific designs described at the register transfer level (RTL) and below. In the context of platform-based design, the trend in the design flow is to raise the level of abstraction of the initial phase and to ground the flow on verified parameterized library modules. Yet, on-chip communications are not supported by a general and formal theory, which is necessary to obtain verified parameterized communication modules. Our utmost objective is to provide a formal foundation to the verification of on-chip communication architectures, spanning from their initial design specifications to their RTL implementation.

As a first step toward this objective, we proposed the generic network on chip model (GeNoC) [7, 8]. It consists of a metamodel of on-chip communication architectures and its implementation in the logic of a theorem proving system. The peculiar aspect of this model is to represent a large class of systems. It is a highly generic and parameterized object. While performing the proofs, parameters such as the

size of the network or the length of messages need not to be instantiated. Its implementation in a theorem proving system provides mechanized support and partial automation in the verification effort. The model is implemented in the ACL2 theorem prover [9]. An important feature of ACL2 is to denote both a powerful theorem proving system and an execution engine in the same environment. The theorem proving system has a high degree of *automation*. ACL2 specifications are written in an applicative subset of Common Lisp and are thus *executable*. These constitute two important advantages of using ACL2.

Our first versions of the model contained unrealistic simplifying hypotheses, concerning the granularity of moves and the time when messages were introduced in the network. As a consequence, while the final result of the communications was correctly portrayed, the intermediate steps were abstracted away: where and when a message is temporarily delayed by the presence of other messages in the network could not be shown.

The enhanced formalization presented in this article constitutes a significant progress, both mathematically simpler and offering a much larger expressive power:

(i) the current metamodel covers a network specification from the *transport* layer to the *data link* layer of the OSI model;

(ii) the progression of the messages in the network is specified step by step instead of considering their transfer atomically from their source to their destination, thus allowing a great variety of scheduling policies (circuit, wormhole, priority, etc.);

(iii) new messages may enter the model at arbitrary times and arbitrary nodes;

(iv) the same model can be used for formal verification and for simulation.

The metamodel detailed in this paper represents the transmission of messages on a *generic* communication architecture, with an *arbitrary* network characterization (topology and node interfaces), routing algorithm, and switching technique. The main function of this model, called GeNoC, is recursive. Each recursive call represents one step of execution, where messages progress by at most one hop. Such a step defines our time unit. A *correctness theorem* is associated with function GeNoC. It states that for all topology $\mathcal{T}$, interfaces $\mathcal{I}$, routing algorithm $\mathcal{R}$, and scheduling policy $\mathcal{S}$ that satisfy associated constraints P_1, P_2, P_3, and P_4, GeNoC fulfills a correctness property $\wp$:

$$\begin{aligned} &\forall \mathcal{T}\ \forall \mathcal{I}\ \forall \mathcal{R}\ \forall \mathcal{S}, \\ &P_1(\mathcal{T}) \wedge P_2(\mathcal{I}) \wedge P_3(\mathcal{R}) \wedge P_4(\mathcal{S}) \\ &\quad \Rightarrow \wp(\mathrm{GeNoC}(\mathcal{T}, \mathcal{I}, \mathcal{R}, \mathcal{S})). \end{aligned} \tag{1}$$

Roughly speaking, the property $\wp$ asserts that *every message arrived at some node n was actually issued at some source node s and originally addressed to node n, and that it reaches its destination without modification of its content*.

The constituents of the metamodel are characterized by constraints P_1, P_2, P_3, and P_4 that express essential properties of the key components, *for example*, wellformedness of the network or termination of the routing function. The proof of formula (1) above is derived from these constraints, without considering the actual definitions of the constituents. Consequently, the global correctness of the network model is preserved *for all particular definitions* satisfying the constraints.

It follows that, for any instance of a network, *that is*, for any $\mathcal{T}_0$, $\mathcal{I}_0$, $\mathcal{R}_0$, and $\mathcal{S}_0$, the property $\wp(\mathrm{GeNoC}(\mathcal{T}_0, \mathcal{I}_0, \mathcal{R}_0, \mathcal{S}_0))$ holds provided that $P_1(\mathcal{T}_0)$, $P_2(\mathcal{I}_0)$, $P_3(\mathcal{R}_0)$, and $P_4(\mathcal{S}_0)$ are satisfied. Hence, verifying formula (1) for a given NoC is reduced to *discharging these instantiated constraints on the NoC constituents*. This verification methodology is illustrated hereinafter on a realistic and state-of-the-art NoC, the Spidergon from STMicroelectronics [10]. (The ACL2 scripts of our model and its instantiation for Spidergon are available at http://tima.imag.fr/vds/GeNoC/genoc.html.)

2. Related Works

Intensive research efforts have been devoted to the development of performance, traffic, or behavior analyzers for NoCs. Most proposed solutions are simulation- or emulation-oriented. Few approaches address the use of (semi)formal methods, essentially toward the detection of faults or debug. A methodology based on temporal assertions is proposed in [11]. It targets a two-level hierarchical ring structure. PSL properties [12] are used to express interface-level requirements and are transformed into synthesizable checkers (monitors). In case of assertion failures, special flits are propagated to a station responsible for analyzing these failures. Goossens et al. [13] advocate communication-centric debug instead of computation-centric debug for complex SoCs and also use a monitor-based solution. They discuss the temporal granularity at which debug can be performed and propose a specific debug architecture with four interconnects. Both approaches rely on a formal expression of properties (assertions), but are indeed founded on a dynamic, nonformal, execution/emulation of a detailed design, where all architectural parameters and implementation decisions have been settled. In contrast, our work applies to the validation of an early specification stage.

Widespread *formal methods* can be classified in two categories: algorithmic techniques (e.g., model-checking) and deductive ones (theorem proving). Many approaches have been proposed in the fields of protocol or network verification in general. Most of them are based on model-checking techniques and target very specific designs. Clarke et al. [14] use the notion of regular languages and abstraction functions to verify temporal properties of families of systems represented by context-free network grammars, for instance, the Dijkstra token ring and a network of binary trees. Creese and Roscoe [15] exploit the inductive structure of branching tree networks and put the emphasis on data independence:

data are abstracted in order to use the FDR model checker to prove properties of CSP specifications. Roychoudhury et al. use the SMV model checker [16] to debug an academic implementation of the AMBA AHB protocol [17]; the model is written at the RTL and without any parameter. Salaün et al. focus on the verification of asynchronous systems using the CADP tool and report the verification of an NoC input controller [18]. Again, the application of model-checking techniques is restricted to architectures where all sizes have been fixed, a later design stage than the one addressed by our model.

Results with theorem provers, or combinations of theorem provers and model-checkers, have also been proposed. Most of them concern specific architectures. The HOL theorem prover [19] is used by Curzon [20] to verify a specific network component, the Fairisle ATM switching fabric. Its structural description is compared to a behavioral specification. Bharadwaj et al. [21] use the combination of the Coq theorem prover [22] and the SPIN model-checker [23] to verify a broadcasting protocol in a binary tree network. Amjad [24] uses a model checker implemented in HOL 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, and HOL is used to verify their composition. In Gebremichael et al. [25], the Æthereal protocol of Philips is specified in the PVS logic. The main property that is verified is the absence of deadlock for an arbitrary number of masters and slaves. The correctness properties checked by these various methods are different from the ones we consider, and the proofs cannot easily be ported from one network structure to another one. We consider this set of works as complementary to ours and applicable to an instantiation of our metamodel to a particular architecture.

The research results that most closely relate to our work tackle the formalization from a *generic* perspective. Moore [26] defines a formal model of asynchrony by a function in the Boyer-Moore logic [27] and shows how to use this general model to verify a biphase mark protocol. More recently, Herzberg and Broy [28] present a formal model of stacked communication protocols, in the sense of the OSI reference model. They define operators and conditions to navigate between protocol layers and consider all OSI layers. This work is more general than Moore 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. In the context of time-triggered architectures, the work of Rushby [29] proposes a general model of time-triggered implementations and their synchronous specifications. The simulation relation between these two models is proven for a large class of algorithms using an axiomatic theory. Pike recently improved the application domain of this theory [30, 31]. Miner et al. [32] define a unified fault-tolerant protocol acting as a generic specification framework that can be instantiated for particular applications. These studies focus on time-triggered protocols. The framework presented in this paper aims at a more general network model and concentrates on the actual interconnect rather than the protocols based on top of this structure.

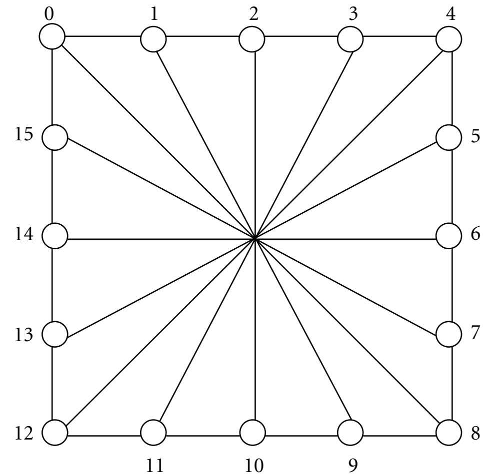

Figure 1: The Spidergon architecture.

3. The Spidergon NoC

The Spidergon NoC from STMicroelectronics [10] is used as a running example throughout this paper. It is an extension of the Octagon network [33] to an arbitrary even number of nodes. Figure 1 gives a 16-node version of the Spidergon. Like the Octagon, it is equipped with a shortest-path routing algorithm.

Spidergon forms a regular architecture, where all nodes are connected to three neighbors and a local IP. Let *NumNode* be its number of nodes. The maximum number of hops is $NumNode/4$ if *NumNode*is a multiple of four. We place our formal representation in that context, we assume a global parameter N such that $NumNode = 4 \cdot N$.

The routing of a message is done as follows. Each node compares the destination address of the message (*DestAd*) to its own address ($NodeAd$) to determine the next action. The node computes the relative address of a message as

$$RelAd = (DestAd - NodeAd) \bmod (4 \cdot N). \quad (2)$$

At each node, the route of the message is a function of *RelAd* as follows.

(i) if $RelAd = 0$, process at node;

(ii) if $0 < RelAd \le N$, route clockwise;

(iii) if $3 \cdot N \le RelAd \le 4 \cdot N$, route counterclockwise;

(iv) route across otherwise.

Example 1. Assume that $N = 4$, that is, $NumNode = 16$. Consider a message M at node 2 sent to node 12. First, $(12 - 2) \bmod 16 = 10$, M is routed across to 10. Then, $(12 - 10) \bmod 16 = 2$, M is routed clockwise to 11, and then to node 12. Finally, $(12 - 12) \bmod 16 = 0$, M has reached its final destination.

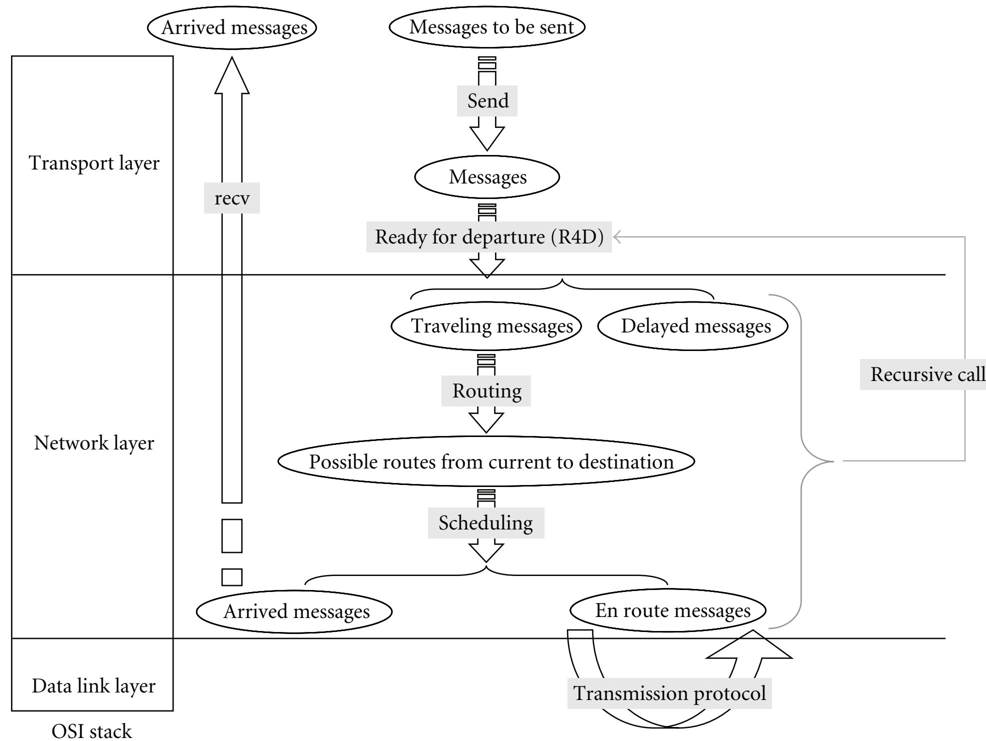

Figure 2: Function GeNoC of the formal model.

We consider a Spidergon network based on the *wormhole switching technique*. Messages are decomposed into smaller units called flits. The *header* flit contains information needed for routing. A *control* flit determines the number of data flits (data payload) that follow in a pipelined fashion.

4. Principles of Our Approach: The Generic Network Model

The main function of the metamodel is function GeNoC sketched in Figure 2. It takes as main argument a list of messages emitted at source nodes and returns a list of messages received at destination nodes. Various intermediate data types are used, which are not described here for the sake of simplicity and clarity. In addition to the data that it is carrying, a *message* contains the following information: its source, destination, current position, the time at which it is injected in the network, and optionally the number of flits in the message. Each message is also uniquely identified by a natural number.

Function 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. It entails the following components:

(i) a *global network state* representing the current status of the memory elements associated with the ports of the nodes and an *explicit notion of time* (Section 5);

(ii) at the *Transport* layer (Section 6), *interfaces* are represented by two functions used to encode messages to be sent in the network (*send*) and to decode them (*recv*). Function *R4D* ("ready for departure") determines which messages can be in the network at the current time. Messages may be injected only at a specific execution time or under constraints on the network load (e.g., credit-based flow control);

(iii) at the *Network* layer (Section 7), the *routing algorithm* is represented by function *Routing* and the *switching technique* is represented by function *Scheduling*. 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. Function *Scheduling* returns a list of messages that have reached their destination and a list of messages that are *en route* to their destination. The model also allows the modeling of *priorities* between messages (e.g., using a round-robin policy);

(iv) at the *data link* layer (Section 8), the *transmission protocol* is specified. The possibility of moving one message from one port to another depends on the current state of the network, which itself depends on the other messages.

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 addresses of its nodes (**A**), a *finite* number of attempts (**att**), the set of arrived messages (**T**, originally empty), the current state of

```
function GeNoC(M, A, att, T, st, z){
if SumOfAtt(att) = 0 then return list (T, M);
else
   begin
   ⟨TR, D⟩ := R4D(M, z);
   ⟨TM, Arr, att', st'⟩ :=
      Scheduling (Routing(TR, A), att, st);
   return GeNoC (union(TM, D), A, att',
                    union(Arr, T), st', z + 1);
   end
}.
```

Algorithm 1

the network (**st**), 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**. The typical recursive form of function GeNoC is as shown in Algorithm 1.

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. First, function *R4D* extracts the messages that are allowed to move at current time **z** (**TR**). For each one of them, routes are computed using function *Routing*. Then, function *Scheduling* computes the list of the arrived messages (**Arr**), the list of messages that are still traveling in the network (**TM**), the remaining attempts (**att**′), and a new state (**st**′). The recursive call processes the traveling messages together with the messages **D** delayed by *R4D*. Time is incremented by 1.

5. Addresses and Network State

5.1. Addresses and Parameters. Networks are assumed to be built from the interconnection of an unbounded, but finite, number of generic switches that have several input and output ports connected to the neighboring switches and to the local IP (Figure 3). Each switch has a *unique* identifier that can be its *position* or *coordinate*.

Tuples composed of a coordinate, a port, and its direction (i.e., tuples of the form $\langle coor, port, dir\rangle$) constitute the basis of the model. We shall refer to such an element as an *address*, abbreviated *a*.

The set of valid addresses is denoted by $\mathcal{A}$. It is computed by a generic function, named *AGen*. The set of valid input arguments to *AGen* is noted $\mathcal{P}_A$. These parameters may represent the dimensions of the network or its number of nodes. The main proof obligation on *AGen* is its typing property.

Proof Obligation 1.

$$AGen : \mathcal{P}_A \longrightarrow \mathcal{A}. \tag{3}$$

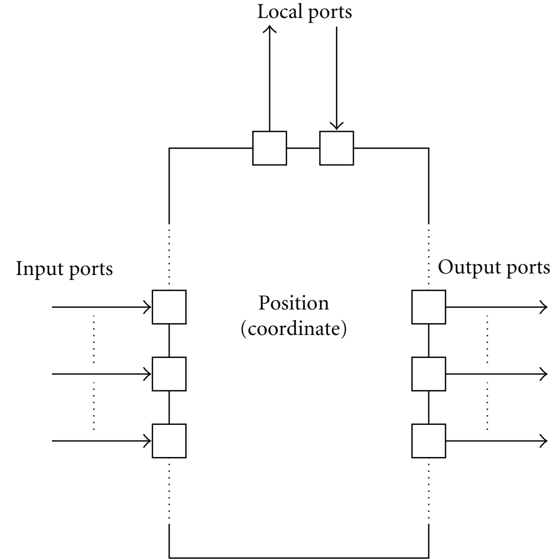

Figure 3: Generic view of switches.

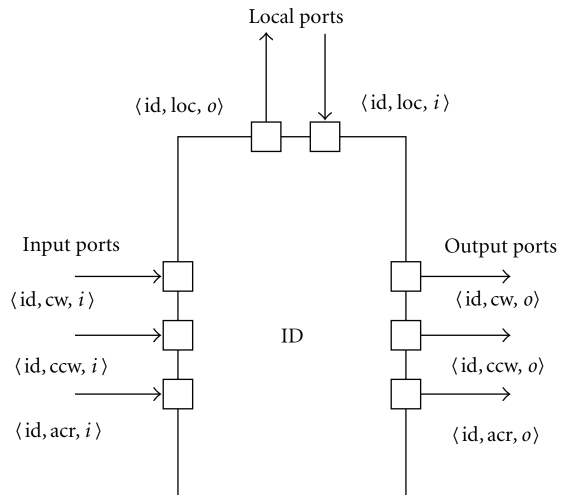

Figure 4: Spidergon switch.

Example 2. In the Spidergon NoC, each switch has four ports: local, clockwise, counterclockwise, and across. The *instantiated* address tuple is $\langle coor, port, dir\rangle$, where *coor* is the switch *id*, *port* belongs to $\{cw, ccw, acr, loc\}$ and *dir* is either "*i*" for input or "*o*" for output (Figure 4).

The instantiation of function *AGen* for the Spidergon network is denoted by *SpiderGen*. It generates the address space of the network. The only parameter of this function is the integer N such that $4 \cdot N$ is the number of nodes in the network. Assume $N = 4$. Then, $\langle 5, cw, i\rangle$ is a member in the address space whereas $\langle 17, ccw, i\rangle$ is not. In general and in our verification effort, the address space of the Spidergon contains $4 \cdot N \cdot 4 \cdot 2$ addresses ($4 \cdot N$ is the number of nodes, 4 is the number of ports per node, and 2 is for the two directions of each port).

The ACL2 verification of Proof Obligation 1 for the Spidergon instantiation is automatic. It involves 4 theorems and 10 functions.

5.2. Network State. A memory element (buffer) is associated to each address, noted *mem*. No assumption is made on that element. If the address has no storage element, *mem* is empty. Otherwise, the structure of this field can be adapted to model the storage elements of a particular instance.

The state of the network consists of a collection of state entries, one for each address. Each state entry is a tuple of the form $\langle a, mem, [opt] \rangle$, where a is the node address, *mem* gives the state of its memory element(s), and $[opt]$ is an optional field that may be used for extra information needed for particular instances (e.g., synchronization signals).

The set of valid states is denoted by $\mathcal{S}$. The set of valid state entries is denoted by $\mathcal{E}$. An entry exists for each element of the address space of a network. Four functions are specified:

(i) *Validntkstate*, recognizes a valid state;

(ii) *StGen*, generates the network initial state;

(iii) *loadBuffer*, loads a message in a buffer;

(iv) *readSEntry*, returns a state entry.

Function *StGen* takes as arguments the parameter used to generate the address space and user-defined parameters, for example, used to dimension the memory elements. The valid set of parameters is denoted by $\mathcal{P}_{st}$. The typing property of function *StGen* constitutes a proof obligation.

Proof Obligation 2.

$$StGen : \mathcal{P}_A \times \mathcal{P}_{st} \longrightarrow \mathcal{S}. \tag{4}$$

Example 3. In our specification of the Spidergon, the addresses generated by function *AGen* (Section 5.1) constitute the address field of state entries. We consider that addresses (or ports) have a one-place buffer. Therefore, the field *mem* is either a message or empty, which is represented by the empty set $\emptyset$. (More precisely, a nonempty buffer contains one message flit. In this section, we write message for brevity.) We consider the Spidergon with a handshake transmission protocol. When transferring data from a port p_s to a port p_d, signal *Rx* is used by p_s to request a transmission, and signal *Ackrx* is used by p_d to acknowledge receipt. The field $[opt]$ of the generic state entry is instantiated by a 2-bit vector representing the values of these two signals for such a port. An example of a valid state entry is $\langle \langle 5, cw, i \rangle, m, 10 \rangle$. This is the state entry of the clockwise input port of node number 5, which contains message m, and where signal *Rx* is 1 and signal *Ackrx* is 0.

The function *loadBuffer* takes as parameters an address, a message, and a network state. It updates the contents of the memory element of the address with the new message. Let $\mathcal{D}_{msg}$ be the domain of messages. The following typing property is a proof obligation for *loadBuffer*.

Proof Obligation 3.

$$loadBuffer : \mathcal{A} \times \mathcal{D}_{msg} \times \mathcal{S} \longrightarrow \mathcal{S}. \tag{5}$$

Example 4. Function $SpiderLoadBuffer(a, m, st)$ updates the field *mem* of the state entry of address a with message m. To empty a buffer, one loads it with $\emptyset$.

The function *readSEntry* takes as parameters an address and a network state. It returns the state entry of the address. The typing property of function *readSEntry* constitutes a proof obligation.

Proof Obligation 4.

$$readSEntry : \mathcal{A} \times \mathcal{S} \longrightarrow \mathcal{E}. \tag{6}$$

Example 5. Function $SpiderReadSEntry(a, st)$ simply returns the state entry with address a.

The most important proof obligation establishes that each state entry corresponds to a valid address ($e.a$ accesses the address field of the state entry).

Proof Obligation 5.

$$\begin{aligned} &p_1 \in \mathcal{P}_A,\ p_2 \in \mathcal{P}_{st}, \\ &e \in StGen(p_1, p_2) \Longleftrightarrow e.a \in AGen(p_1). \end{aligned} \tag{7}$$

Example 6. The ACL2 verification of this state specification for Spidergon is largely automatic, only two additional lemmas are needed.

6. Transport Layer

This layer is responsible for encapsulating data into packets suitable for transfer to the network infrastructure or managing the reverse transaction for delivering messages to the destination node. Our model specifies this encoding/decoding process as well as the flow control services of the transport layer.

6.1. Interfaces. The interfaces are modeled by two functions. Function *send* represents message encoding, and function *recv* specifies decoding. To guarantee the compatibility between these two functions, a proof obligation states that their composition is the identity function.

Proof Obligation 6.

$$\forall\, msg \in \mathcal{D}_{msg}, \quad (recv \circ send)(msg) = msg. \tag{8}$$

6.2. Network Access Control. The network access control function checks whether a message can be injected into the network. This is modeled by function *R4D*. This function splits a list l of messages into the list *R4D*.traveling(l) of messages that satisfy the departure conditions, and the list *R4D*.delayed(l) of delayed messages. (This notation is used

to represent the constituent of $R4D(l)$ that contains the traveling messages. Similar notations are used throughout the rest of this paper) . The three lists belong to $\mathcal{D}_{lm}$, which denotes the set of all possible lists of messages.

We can consider various access control conditions. Example conditions are credits allowed for each node to send messages (e.g., in the Faust network [3]), or a maximum number of messages allowed in the network (e.g., in the Nostrum network [34]).

Function *R4D* takes as arguments a list of messages and a user-defined parameter used to model particular departing conditions. We use "$*$" to represent the arbitrary domain of this parameter. The typing property of function *R4D* constitutes a proof obligation.

Proof Obligation 7.

$$R4D : \mathcal{D}_{lm} \times * \longrightarrow \mathcal{D}_{lm} \times \mathcal{D}_{lm}. \tag{9}$$

Each one of the outputs of function *R4D* must be a subset of the input message list.

Proof Obligation 8.

$$R4D.delayed(l) \subseteq l \wedge R4D.traveling(l) \subseteq l. \tag{10}$$

A message cannot be in both lists at the same time, that is, the intersection of these two lists must be the empty set.

Proof Obligation 9.

$$R4D.delayed(l) \bigcap R4D.traveling(l) = \varnothing. \tag{11}$$

Example 7. The instance of function *R4D* for the Spidergon is called *SpiderR4D*.

We are not aware of any flow control mechanism for the Spidergon network. Only the message injection time (simulation step) is considered in *SpiderR4D*. Each message can depart if its injection time is equal to the current simulation step. In that case, it is put in list *SpiderR4D.*traveling. Otherwise, it is added to list *SpiderR4D.*delayed. Once put in list *SpiderR4D.*traveling, it becomes a traveling message, and it will remain in this list in the recursive calls.

All the proof obligations have been discharged for function *SpiderR4D*. Proof Obligation 8 needed 2 intermediate lemmas for a total proof time of 2.41 seconds. As for Proof Obligation 9, one intermediate lemma is needed for a proof time of 1.45 seconds.

7. Network Layer

The Network layer is responsible for the end-to-end (i.e., source to destination) message delivery. In particular, it implements the routing method that determines the intermediate nodes from the source to the destination. The congestion and conflict management of the switching technique is also a Network layer responsibility [35].

7.1. Routing. Our representation of the routing algorithm involves the main function *Routing* that computes all possible routes from the current position to the destination for all traveling messages, as well as two auxiliary functions:

(i) *RCore*, computes all possible routes from the current position to the destination for one message;

(ii) *RLogic*, implements the routing logic, that is, computes the next step node.

In case of deterministic algorithms like the Spidergon routing algorithm, there is only one possible route. Several routes are feasible when adaptive algorithms are considered.

Function *RCore* simply applies *RLogic* recursively. Function *Routing* also recursively applies *RCore* to all messages. Function *RLogic* takes two addresses *curr* and *dest* as parameters and computes the next hop on the route from *curr* to *dest*. These three nodes must belong to $\mathcal{A}$.

Proof Obligation 10.

$$RLogic : \mathcal{A} \times \mathcal{A} \longrightarrow \mathcal{A}. \tag{12}$$

Example 8. The routing logic determines the possible moves between two adjacent addresses. The Spidergon routing logic offers four possible moves: clockwise, counterclockwise, across, and to the local IP. These four moves are represented by four functions. For instance, the function for a clockwise move is as shown in Algorithm 2.

The routing logic, represented by function *SpiderRLogic*, is then defined from these four moves following the Spidergon routing algorithm described in Section 3 as shown in Algorithm 3.

This function constitutes a valid instance of function *RLogic* and satisfies the above proof obligation.

Function *RCore* takes as parameters two addresses: the current one and the destination. In case of deterministic algorithms, it computes the route from the current node to the destination by recursive applications of function *RLogic*. The set of all possible routes in the network is denoted by $\mathcal{R}$. Function *RCore* returns an element of $\mathcal{R}$.

Proof Obligation 11.

$$RCore : \mathcal{A} \times \mathcal{A} \longrightarrow \mathcal{R}. \tag{13}$$

Example 9. Function *SpiderRCore* recursively calls function *SpiderRLogic*, as shown in Algorithm 4. This function satisfies the above proof obligation.

A route r for a message m is correct if its first element is the current position of m and its last element is the destination of m. All addresses of a route must belong to the address space. Predicate *ValidRoutep* checks the conjunction of these conditions ($m.curr$, and $m.dest$ represent the current position and destination of m):

$$\begin{aligned} & ValidRoutep(m, r, \mathcal{A}) \\ & \quad \triangleq first(r) = m.curr \wedge last(r) = m.dest \wedge r \subseteq \mathcal{A}. \end{aligned} \tag{14}$$

```
function move-clockwise(c) {
if c.dir = i then    // internal hop to output port
    return ⟨c.id, cw, o⟩;
else    // leave from port ⟨c.id, cw, o⟩
    return ⟨(c.id + 1) mod 4 · N, ccw, i⟩;
        // enter on port to ⟨c.id + 1, ccw, i⟩ of neighbor
}
```

Algorithm 2

```
function SpiderRLogic(curr, dest) {
RelAd := (dest.id − curr.id) mod 4N;
if RelAd = 0 then return move-local(curr);
else
    if 0 < RelAd ≤ N then
     return move-clockwise(curr);
    else
        if 3 · N ≤ RelAd ≤ 4 · N then
            return move-counterclockwise(curr);
        else return move-across(curr);
}.
```

Algorithm 3

Function *Routing* takes as arguments a list of messages and the address space. It returns a list of *routed messages*, that is, messages associated with routes. $\mathcal{D}_{lm}$ denotes the set of lists of messages, and $\mathcal{D}_{lrm}$ denotes the set of lists of routed messages.

Proof Obligation 12.

$$Routing : \mathcal{D}_{lm} \times \mathcal{D}_{\mathcal{A}} \longrightarrow \mathcal{D}_{lrm}. \tag{15}$$

Example 10. Function *Routing* is instantiated by function *SpiderRouting* as shown in Algorithm 5.

Functions *first* and *tail* return the first element of a list and the tail of the list (i.e., the list without its first element).

The well-formedness of the routes produced by function *Routing* is expressed by predicate *CorrectRoutesp*. This predicate takes as arguments a list of messages *lm*, the list of the corresponding routed messages *lrm*, and the address space $\mathcal{A}$. It verifies that, for every message *m* of *lrm*, there exists a unique routed message *rm* in *lrm* such that *m* and *rm* have the same *id* and that the route of *rm* is correct with respect to message *m*.

$$CorrectRoutesp(lm, lrm, \mathcal{A}) \triangleq \forall\, m \in lm, \exists rm \in lrm,$$
$$ValidRoutep(m, rm.route, \mathcal{A}) \land m.id = rm.id. \tag{16}$$

The main proof obligation guarantees that all routes produced by function *Routing* are correct.

```
function SpiderRCore (c, d) {
if c = d then return d;    // at destination
else    // do one hop
    return list(c, SpiderRCore(SpiderRLogic(c, d), d));
}.
```

Algorithm 4

```
function SpiderRouting (lm, nodeset) {
if lm = ∅ then return ∅;
else
    m := first(lm);
    route := SpiderRCore(m.curr, m.dest);
    rm := ⟨m, route⟩;    // routed message
    return list(rm, SpiderRouting (tail(lm), nodeset));
}.
```

Algorithm 5

Proof Obligation 13.

$$CorrectRoutesp(lm, Routing(lm, \mathcal{A}), \mathcal{A}). \tag{17}$$

Example 11. We first prove that function *SpiderRCore* produces valid routes, that is, its result satisfies predicate *ValidRoutep*. The proof proceeds by induction on the length of a route. The remaining properties are trivial. Overall, the specification and the validation of the routing algorithm of Spidergon amount to 1200 lines of ACL2 code. Most of the difficulty comes from arithmetic reasoning in ACL2.

7.2. Scheduling. Function *Scheduling* determines the progression of messages in the network. From a list of messages, it returns the list of messages that have reached their destination and the list of messages that are still *en route* to their destination. *En route* messages may be blocked on their current node or able to move forward. Two layers of the OSI stack are concerned here

(i) At the Network layer, priorities are used to resolve the competition for the next resource (e.g., a port). Their specifications are discussed in Section 7.3. Other aspects, specific to a switching technique, can also be implemented at this layer. For example, in wormhole switching, flits belonging to the same message must follow the same route in a pipelined fashion, without the introduction of flits belonging to other messages between them [36].

(ii) At the data link layer, data transmission follows a low-level protocol that determines whether the next resource is ready to receive. This is the purpose of Section 8.

Function *Scheduling* takes as arguments a list of routed messages, the current value of the attempts (of type $\mathcal{D}_{att}$), the current network state, and an additional parameter that

models the priority order to be used. This parameter is user defined and we use an "$*$" to denote its domain. Function *Scheduling* returns two lists (*en route* messages used in the recursive call of GeNoC, and arrived messages), the updated attempts, and the updated network state.

Proof Obligation 14.

$$\begin{aligned} &Scheduling: \\ &\mathcal{D}_{lrm} \times \mathcal{D}_{att} \times \mathcal{S} \times * \longrightarrow \mathcal{D}_{lm} \times \mathcal{D}_{lrm} \times \mathcal{D}_{att} \times \mathcal{S}. \end{aligned} \tag{18}$$

Example 12. *WHS* is the core function in modeling the wormhole switching technique. It takes as arguments a list of routed messages, three accumulators (initially empty) and the global state. The first two accumulators collect the messages that are still *en route* (*ER*) and those that have reached their destination (*Arr*). The last accumulator (*2Bmoved*) collects the ports from which a message moves: their buffer must be emptied (see function *WHScheduling* thereafter). *WHS* returns the three accumulators and a new state. If a move is possible (lines 10 to 23), the message is added to *2Bmoved* (lines 14 and 21). The memory element of the target of the hop (second element of the route) is updated with the content of the message (lines 15 and 22). If the message has reached its destination, it is added to *Arr* (line 13). Otherwise, the first address of its route is removed—the second address becomes its current position simulating the action of making a hop—and the message is then added to *ER* (line 20). If no move is possible, the message is simply added to *ER* without any modification (lines 26 to 28) (see Algorithm 6).

Function *WHScheduling* is the top level function in the model of the scheduling policy and the instance of function *Scheduling*. It includes the priority ordering between the ports of a node and the low-level protocols used in the data link layer. Function *WHScheduling* first calls these functions to order its input argument (line 2) and to compute the first phase of our handshake protocol (line 3). Function *rrSort* implements a round-robin priority scheme. Function *WHS* is called with empty accumulators (only represented as "..." in the definition).

To build the returned state, *WHScheduling* empties the buffer of the addresses that were left by a message (line 9), produced by function *WHS* above in accumulator *2Bmoved*. Function *free* simply loads all these addresses with $\varnothing$, the empty buffer as shown in Algorithm 7.

```
1: function WHS(lrm, ER, Arr, 2Bmoved, st) {
2: if lrm = ∅ then
3:    return list(ER, Arr, 2Bmoved, st);
4: else
5:    begin
6:    rm := first(lrm);
7:    r := SpiderTestRoutes.r(st, rm);
8:    newst := SpiderTestRoutes.st(st, rm);
              // see Section 8
9:    end
10: if r ≠ ∅ then    // a hop is possible for rm
11:    if len(r) = 2 then    // rm reached destination
12:       begin
13:       Arr := insert(rm, Arr);
14:       2Bmoved := insert(rm, 2Bmoved);
15:       st' :=
             SpiderLoadBuffer(Second(r), rm, st);
16:       return    // recursive call
          WHS(tail(lrm), ER, Arr, 2Bmoved, st');
17:       end
18:    else    // rm still en route
19:       begin
20:       ER := insert(hop(rm), ER);
21:       2Bmoved := insert(rm, 2Bmoved);
22:       st' :=
             SpiderLoadBuffer(Second(r), rm, st);
23:       return    // recursive call
          WHS(tail(lrm), ER, Arr, 2Bmoved, st');
24:       end
25: else    // no move
26:    begin
27:    ER := insert(rm, ER);
28:    return    // recursive call
          WHS(tail(lrm), ER, Arr, 2Bmoved, st);
29:    end
30: }.
```

Algorithm 6

The main property required for the arrived messages is that their routes are left unchanged. Predicate *ArrivedCorrp* below checks that for every routed message rm' of the list lrm', there exists a routed message *rm* of the list *lrm* such that *rm* and rm' have the same *id* and the same routes *r*.

$$\begin{aligned} &ArrivedCorrp(lrm, lrm') \triangleq \forall rm' \in lrm', \exists rm \in lrm, \\ &rm.id = rm'.id \land rm.r = rm'.r. \end{aligned} \tag{19}$$

The following proof obligation ensures that the arrived messages satisfy this predicate.

Proof Obligation 15.

$$ArrivedCorrp(Scheduling.arr(lrm, a, s, o), lrm). \tag{20}$$

Similarly, predicate *EnrouteCorrp* is designed for the correctness of the *enRoute* messages and used in the next proof obligation. Formally, it checks that for every *enRoute* message, there exists a routed message of the input list of *Scheduling* which has the same *id*, origin, injection time, and so forth.

Proof Obligation 16.

$$EnrouteCorrp(Scheduling.enRoute(lrm, a, s, o), lrm). \tag{21}$$

The next proof obligation states that the two output lists of function *Scheduling* are subsets of the parameter *lrm*.

```
1: function WHScheduling(lrm, att, st, o) {
2: lrm := rrSort(lrm, o);    // see Section 7.3
3: rst := SpiderReqTrans(st);    // see Section 8
4: EnRoute := WHS.EnRoute(lrm, . . . , rst);
5: Arrived := WHS.Arrived(lrm, . . . , rst);
6: 2Bmoved := WHS.2Bmoved(lrm, . . . , rst);
7: att' := ConsumeAttempts(att);
8: st' := WHS.st(lrm, . . . , rst);
9: st'' := free(2Bmoved, st');
10: return list(EnRoute, Arrived, att', st'');
11: }.
```

Algorithm 7

Proof Obligation 17.

$$\begin{aligned} & Scheduling.arr(lrm, a, s, o) \subseteq lrm \\ & \wedge\ Scheduling.enRoute(lrm, a, s, o) \subseteq lrm. \end{aligned} \tag{22}$$

The last proof obligation states that a message cannot be in both lists at the same time, that is, the intersection of these two list is the empty set.

Proof Obligation 18.

$$\begin{aligned} & Scheduling.arr(lrm, a, s, o) \\ & \bigcap Scheduling.enRoute(lrm, a, s, o) = \varnothing. \end{aligned} \tag{23}$$

Example 13. For the whole wormhole instance, 17 functions and 60 lemmas are defined. The detailed figures for the successive proof obligations (PO) are

PO 15: 4 lemmas, total proof time of 4 seconds;

PO 16: 21 lemmas, total proof time of 47 seconds;

PO 17: 5 lemmas, total proof time of 0.61 seconds;

PO 18: 2 lemmas and largely automatic, 1.18 seconds.

7.3. Priorities. Function *PrioSort* is used to model the priority scheme at the Network layer. It takes as parameter a list of messages and sorts it according to the user-defined parameter passed to function *Scheduling*. The priority policy can be based on a round-robin rule such as in the Hermes NoC [37], determined by the "age" of the messages like in Nostrum [34], or any other ordering. The signature of *PrioSort* is given by the following proof obligation.

Proof Obligation 19.

$$PrioSort : \mathcal{D}_{lm} \times * \longrightarrow \mathcal{D}_{lm}. \tag{24}$$

Example 14. We instantiate the priority scheme with a round-robin policy, implemented in function *rrSort*. The user-defined parameter is simply the initial ordering at a node. At each simulation step, we permute this ordering. The initial ordering is *loc*, *cw*, *ccw*, *acr*, meaning that the local port has the priority. At every step, this ordering is shifted starting at the position of the port that has been served in the current step. For instance, if a node performs a read or a write to port *ccw*, the new ordering will be *acr*, *loc*, *cw*, *ccw*.

Another proof obligation guarantees that *PrioSort* is actually a sorting function, *that is*, its result is a permutation of its parameter lm. $IsPerm(l_1, l_2)$ is the predicate that checks whether l_1 is a permutation of l_2.

Proof Obligation 20.

$$IsPerm(PrioSort(lm, o), lm). \tag{25}$$

Example 15. The instance of the sorting function used in the Spidergon network needed 270 lines of code. Proof Obligations 19 and 20 required 13 common lemmas, proven in 6.92 seconds. One extra intermediate lemma was necessary for Proof Obligation 19, for a total time of 0.3 seconds. Proof Obligation 20 was verified in 8.6 seconds.

8. Data Link Layer

The data link layer is responsible for the point-to-point transmission between two adjacent nodes. In particular, it implements the transmission protocol. Various types of protocols exist, such as simple handshake protocols [37] or sophisticated algorithms based, for instance, on load analysis [34].

The main function of our representation is *TestRoutes* that checks the feasibility of the next hop on the route. As mentioned in Section 5.2, an optional field can be added to the network state, for instance, to specify the request and acknowledge signals of a handshake protocol. In that case, the protocol is represented using two auxiliary functions:

(i) *ReqTrans*, expresses the transmission of requests;

(ii) *AckTrans*, expresses the transmission of acknowledgments.

Function *ReqTrans* updates the optional fields of the current state by setting request signals to 1. It takes the current state as parameter and returns the updated state.

Proof Obligation 21.

$$ReqTrans : \mathcal{S} \longrightarrow \mathcal{S}. \tag{26}$$

Example 16. This function is instantiated by function *SpiderReqTrans*. Every port with a nonempty buffer requests a transfer (we have one-place buffers, thus nonemptiness means that a flit has to be transferred). Function *ActivReq* sets the *Rx* signal of state element *e* to 1 as shown in Algorithm 8.

This function statisfies the above proof abligation.

Function *AckTrans* takes as parameters the current network state and an address. It returns a new state in which the acknowledgment signal of this address is set to 1 if transmission to this address is possible.

```
function SpiderReqTrans(st) {
e := first(st);
if e.mem = ∅ then     // no request
    return list(e, SpiderReqTrans(tail(st)));
else // request issued
return list(ActivReq(e),SpiderReqTrans(tail(st)));
}.
```

Algorithm 8

```
function SpiderAckTrans(st, a) {
if SpiderReadSEntry(st, a).mem = ∅ then
    return ActivAck(st, a);
else return st;
}
```

Algorithm 9

Proof Obligation 22.

$$AckTrans : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{S}. \tag{27}$$

Example 17. This function is instantiated by function *SpiderAckTrans*, which sets the *Ackrx* signal to 1 if the target address has an empty buffer. This is done by function *ActivAck* (see Algorithm 9).

Function *TestRoutes* takes as parameters a state and a routed message (of type $\mathcal{D}_{rm}$). It returns a new state and a route. If the route is empty, it means that no route is possible for the message, that will temporarily remain blocked on its current node. Otherwise, the next hop on the route is valid for the message.

Proof Obligation 23.

$$TestRoutes : \mathcal{S} \times \mathcal{D}_{rm} \longrightarrow \mathcal{S} \times \mathcal{R} \bigcup \{\varnothing\}. \tag{28}$$

Example 18. Function *SpiderTestRoutes* is the valid Spidergon instance of function *TestRoutes* (see Algorithm 10).

The instance for the handshake protocol needed 80 lines of ACL2 code.

In the case of *adaptive* algorithms, several routes would be possible and the instance of function *TestRoutes* would check all possible routes.

Table 1 summarizes the proof effort for the generic model and for the Spidergon instantiation. Such instantiations can be performed by an experienced ACL2 user; a good understanding of the principles and of the proof obligations of the generic model is necessary. All proof times are measured on an Intel Core Duo T2400, 1.83 GHz, with 1.5 GBs of RAM.

```
function SpiderTestRoutes(st, rm) {
r := rm.route;
nxthop := second(r);    // possible next hop
st' := SpiderAckTrans(st, nxthop);
if st' ≠ st then
    return (st', r);
else
    return (st, ∅);
}.
```

Algorithm 10

```
function SpiderGeNoC(M, A, att, T, st, z) {
if SumOfAtt(att) = 0 then return list (T, M);
else
    begin
    ⟨TR, D⟩ := SpiderR4D(M, z);
    ⟨TM, Arr, att', st'⟩ :=
     WHScheduling(SpiderRouting(TR, A), att, st);
    return
     SpiderGeNoC
     (union(TM, D), A, att', union(Arr, T), st', z + 1);
    end
}.
```

Algorithm 11

9. GeNoC Instance—Proof—Simulation

9.1. Formal Proof. In the introduction, we sketched the correctness theorem (1) associated with function GeNoC that amounts to verifying the property $\wp$: *every message arrived at some node n was actually issued at some source node s and originally addressed to node n, and the message reaches its destination without modification of its content.*

In other words, for each arrived message (i.e., each element of GeNoC.$T(M, A, att, T, st, z)$), there exists a message in M with the same *id*, the same destination (*dest*), and the same content, denoted by frm. The corresponding formalization is as follows.

Theorem 1.

$$\begin{aligned}\forall m \in GeNoC.T(M, A, att, T, st, z),\ \exists m' \in M,\\ m.id = m'.id \wedge m.dest = m'.dest\\ \wedge m.frm = m'.frm.\end{aligned} \tag{29}$$

Example 19. The GeNoC instance of the Spidergon network is given in Algorithm 11. It follows the pattern of Section 4, with the Spidergon instances *SpiderR4D*, *SpiderRouting*, and *WHScheduling*.

Once the different proof obligations of each component have been proven, the general correctness property (the theorem above) is directly obtained from the instantiated proof obligations. No additional verification effort is required.

Table 1: Proofs-size and CPU time.

Proof	Number of functions	Number of theorems	CPU time
Generic model	114	239	86.96 s
Spidergon			
Topology (105 ACL2 lines)	10	4	1.23 s
Network state (300 ACL2 lines)	15	10	0.3 s
Access control (220 ACL2 lines)	3	23	6.51 s
Routing (1200 ACL2 lines)	39	90	2092.81 s
Scheduling (2100 ACL2 lines)	17	60	75 s
Priority schemes (270 ACL2 lines)	7	22	17.75 s
Handshake protocol (80 ACL2 lines)	6	3	0.16 s

Table 2: Simulation example.

Id	Source	Content	Destination	Flits	Time
1	(0 Loc i)	(11 12)	(8 Loc o)	4	1
2	(1 Loc i)	(21 22 23)	(8 Loc o)	5	0
3	(4 Loc i)	(31)	(3 Loc o)	3	2
4	(5 Loc i)	(41 42)	(3 Loc o)	4	0

Table 3: Routes of the messages.

Id	Route
1	(2, (0 Loc i)) (3, (0 Acr o)) (4, (8 Acr i)) (5, (8 Loc o))
2	(1, (1 Loc i)) (2, (1 Acr o)) (3, (9 Acr i)) (4, (9 Ccw o)) (5, (8 Cw i)) (10, (8 Loc o))
3	(3, (4 Loc i)) (4, (4 Ccw o)) (5, (3 Cw i)) (6, (3 Loc o))
4	(1, (5 Loc i)) (2, (5 Ccw o)) (3, (4 Cw i)) (8, (4 Ccw o)) (9, (3 Cw i)) (10, (3 Loc o))

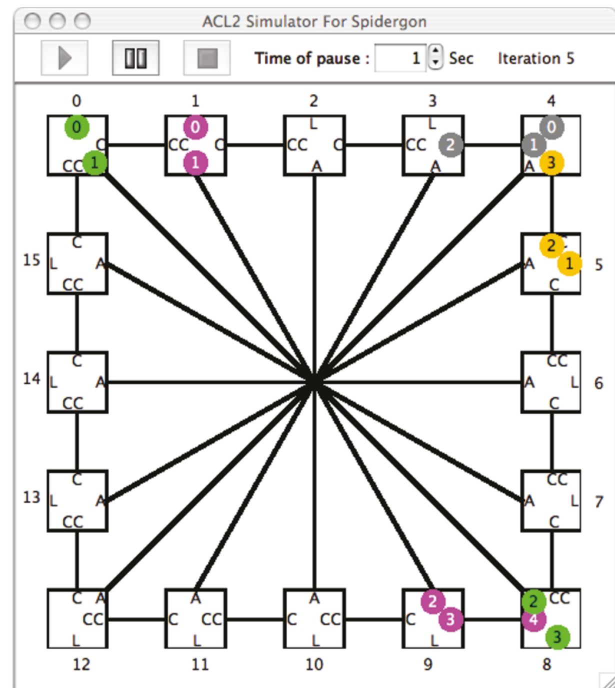

Figure 5: Animated ACL2 simulation.

9.2. Simulation. We mentioned in the introduction that ACL2 provides both a theorem proving system and an execution engine. With the ACL2 implementation of our model, we not only perform formal proofs, but we also can execute the NoC behavior on test cases. A real synergy is thus obtained. While global safety properties can be demonstrated with theorem proving techniques, the detailed analysis of message progression and interaction in the network still requires the simulation of test scenarios. It is of utmost importance that these two verification techniques be applicable to the *same* model. Our model enables this synergy.

We now illustrate this characteristic with a simple simulation of the Spidergon. In the simulation scenario described by Table 2, four messages are considered. Message 1 is split into 2 flits (hence its total number of flits is 4 due to the presence of the header flit and of the *number of flits* one), message 2 is split into 3 flits (its total number of flits is 5), and messages 3 and 4 occupy 3 and 4 flits.

The ACL2 execution of GeNoC yields 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. Figure 5 gives a screenshot of the 5th simulation step. Flits are numbered in decreasing order, from *Nbflits-1* downto 0.

Messages 1 and 2 on the one hand and 3 and 4 on the other hand have the same destination. The messages follow the routes given in Table 3, where each location is associated with the iteration cycle (iteration 1 corresponds to time 0).

Message 1 departs at iteration cycle 2 and reaches the local port of node 8 before message 2, thus blocking it on port *cw* of node 8, as shown on the screenshot, until step 10. Message 4 arrives on the *ccw* port of node 4 when the first flit of message 3 departs on the local port of this node. The round-robin rule gives the priority to message 3, and message 4 completes its travel to its destination as soon as this node is freed (step 8). This simulation is faithful to the expected behavior.

10. Conclusion

In this paper, we presented a formal metamodel for reasoning about network specifications, from the transport to the data link layer of the OSI architectural model. In the metamodel, the main constituents of an NoC architecture—(1) topology, (2) interfaces, (3) routing, (4) scheduling—are characterized by their essential properties rather than their size or detailed behavior. Reasoning is based on the properties in order to prove safety correctness statements about the overall network architecture. In the metamodel, it is proved once and for all that the characteristic properties of the components imply the correctness statement.

To instantiate the metamodel on a particular NoC architecture, one has to fix (1) the node interconnection and how addresses are formed, (2) the interfaces and state holding elements of the nodes, (3) the progression of messages between neighboring nodes and the corresponding state transition functions, and (4) the priority or conflict resolution policy. Each one of these decisions is documented as a high level, executable algorithmic specification for the corresponding NoC constituent. For each one of them, all that needs to be done is the proof that the essential properties hold on the algorithmic specification. This is a modular proof, each constituent is processed in isolation. The number of ancillary definitions and intermediate lemmas, the size of the proof script, and the execution time of the proof system are reasonable.

Once all the proof obligations for the four constituents have been discharged, the overall correctness statement holds on the model of the NoC architecture as an immediate consequence of the proof performed on the metamodel. What has been obtained at this point is a high-level correctness statement for a parameterized design. All reasoning has been made irrespective of the message/flit size, number of nodes, link width, or other magnitude parameters, all of which will be set at a later design stage for a particular implementation of the NoC in a particular circuit.

In our work, the metamodel and its instantiations are coded in Lisp and the verification flow is supported by the ACL2 proof system. This is a beneficial feature for the credibility of our work seen from the point of view of the NoC designer. The same model that is used for proof is also executable and can be run dynamically on numeric test cases. In a previous case study, the Hermes NoC, for which a VHDL design was available to us, we could show the exact same results on the same test scenarios between the VHDL simulation and the GeNoC execution.

The GeNoC metamodel has been instantiated on a variety of network definitions (Octagon [33], Hermes [37], Spidergon [10]), and for each topology on the various routing and scheduling policies considered by their designers. We are now confident that the current metamodel has reached a sufficient level of expressiveness to support the usual message forwarding on regular structures. The correctness statement discussed in this paper is just one of the many safety properties that must be established; its merit is twofold. We believe that we are first to have formally proved it, and it has served the purpose of demonstrating our approach and proof process. We are now turning our attention to formalizing the absence of deadlocks and livelocks in the metamodel.

In terms of formal verification platform for NoCs, GeNoC constitutes the most abstract layer in the verification flow. The next step will be the refinement of the metamodel to support register transfer level. We want to provide a verification method, supported by proof tools, to show that a cycle accurate RTL model correctly implements a GeNoC instantiation and propagate in the process the satisfaction of the proof obligations. This is the direction of our research in the near future.

References

[1] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[2] P. P. Pande, C. Grecu, A. Ivanov, R. Saleh, and G. De Micheli, "Design, synthesis, and test of networks on chips," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 404–413, 2005.

[3] F. Clermidy, D. Varreau, and D. Lattard, "A NoC-based communication framework for seamless IP integration in complex systems," in *Proceedings of the International Workshop on IP-Based System-on-Chip Design (IPSoC '05)*, Grenoble, France, December 2005.

[4] D. Wiklund and D. Liu, "SoCBUS: switched network on chip for hard real time embedded systems," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, pp. 78–85, Nice, France, April 2003.

[5] A. Sangiovanni-Vincentelli, "Defining platform-based design," *EEDesign of EETimes*, February 2002.

[6] R. Dubey, "Elements of verification," *SOCcentral*, March 2005.

[7] J. Schmaltz and D. Borrione, "A functional formalization of on chip communications," *Formal Aspects of Computing*, vol. 20, no. 3, pp. 241–258, 2008.

[8] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz, "A generic model for formally verifying NoC communication architectures: a case study," in *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS '07)*, pp. 127–136, Princeton, NJ, USA, May 2007.

[9] M. Kaufmann, P. Manolios, and J. Strother Moore, *Computer Aided Reasoning: An Approach*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[10] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon: a novel on-chip communication network," in *Proceedings of the International Symposium on System-on-Chip*, pp. 1–15, Tampere, Finland, November 2004.

[11] J. S. Chenard, S. Bourduas, N. Azuelos, M. Boulé, and Z. Zilic, "Hardware assertion checkers in online detection of network-on-chip faults," in *Proceedings of the Workshop on Diagnostic Services in Networks-on-Chips*, Nice, France, April 2007.

[12] IEEE Std 1850-2005, "IEEE Standard for Property Specification Language (PSL)," IEEE, 2005.

[13] K. Goossens, B. Vermeulen, R. van Steeden, and M. Bennebroek, "Transaction-based communication-centric debug," in *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS '07)*, pp. 95–106, Princeton, NJ, USA, May 2007.

[14] E. M. Clarke, O. Grumberg, and S. Jha, "Verifying parameterized networks," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 5, pp. 726–750, 1997.

[15] S. Creese and A. Roscoe, "Formal verification of arbitrary network topologies," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, vol. 2, pp. 1033–1039, Las Vegas, Nev, USA, June-July 1999.

[16] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Press, Dordrecht, The Netherlands, 1993.

[17] A. Roychoudhury, T. Mitra, and S. R. Karri, "Using formal techniques to debug the AMBA system-on-chip bus protocol," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 828–833, Munich, Germany, March 2003.

[18] G. Salaün, W. Serwe, Y. Thonnart, and P. Vivet, "Formal verification of CHP specifications with CADP illustration on an asynchronous network-on-chip," in *Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '07)*, pp. 73–82, Berkeley, Calif, USA, March 2007.

[19] M. Gordon and T. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, Cambridge, UK, 1993.

[20] P. Curzon, "Experiences formally verifying a network component," in *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS '94)*, pp. 183–193, IEEE Press, Gaithersburg, Md, USA, June 1994.

[21] R. Bharadwaj, A. Felty, and F. Stomp, "Formalizing inductive proofs of network algorithms," in *Proceedings of the Asian Computing Science Conference on Algorithms, Concurrency and Knowledge (ACSC '95)*, pp. 335–349, Pathumthani, Thailand, December 1995.

[22] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*, Springer, Berlin, Germany, 2004.

[23] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[24] H. Amjad, "Model checking the AMBA protocol in HOL," Tech. Rep., Computer Laboratory, University of Cambridge, Cambridge, UK, September 2004.

[25] B. Gebremichael, F. W. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu, "Deadlock prevention in the Æthereal protocol," in *Proceedings of the 13th IFIP WG 10.5 Advanced Research Working Conference Correct Hardware Design and Verification Methods (CHARME '05)*, pp. 345–348, Saarbrücken, Germany, October 2005.

[26] J. Strother Moore, "A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol," *Formal Aspects of Computing*, vol. 6, no. 1, pp. 60–91, 1994.

[27] R. S. Boyer and J. Strother Moore, *A Computation Logic Handbook*, Academic Press, New York, NY, USA, 1988.

[28] D. Herzberg and M. Broy, "Modeling layered distributed communication systems," *Formal Aspects of Computing*, vol. 17, no. 1, pp. 1–18, 2005.

[29] J. Rushby, "Systematic formal verification for fault-tolerant time-triggered algorithms," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 651–660, 1999.

[30] L. Pike, "A note on inconsistent axioms in Rushby's "systematic formal verification for fault-tolerant time-triggered algorithms"," *IEEE Transactions on Software Engineering*, vol. 32, no. 5, pp. 347–348, 2006.

[31] L. Pike, "Modeling time-triggered protocols and verifying their real-time schedules," in *Proceedings of Formal Methods in Computer Aided Design (FMCAD '07)*, pp. 231–238, Austin, Tex, USA, November 2007.

[32] P. S. Miner, A. Geser, L. Pike, and J. Maddalon, "A unified fault-tolerance protocol," in *Proceedings of the Joint International Conferences on Formal Modeling and Analysis of Timed Systmes and Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '04)*, Y. Lakhnech and S. Yovine, Eds., vol. 3253 of *Lecture Notes in Computer Science*, pp. 167–182, Springer, Grenoble, France, September 2004.

[33] F. Karim, A. Nguyen, and S. Dey, "An interconnect architecture for networking systems on chips," *IEEE Micro*, vol. 22, no. 5, pp. 36–45, 2002.

[34] A. Jantsch, "Models of computation for networks on chip," in *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD '06)*, pp. 165–176, IEEE Computer Society, Turku, Finland, June 2006.

[35] A. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, USA, 2002.

[36] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, San Francisco, Calif, USA, 2003.

[37] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "HERMES: an infrastructure for low area overhead packet-switching networks on chip," *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 69–93, 2004.