Towards A Formally Verified Network-on-Chip

Tom van den Broek and Julien Schmaltz
Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
Email: tombroek@science.ru.nl,julien@cs.ru.nl

***Abstract*—Multi-Processor Systems-on-Chip (MPSoC) designs are constructed by assembling pre-designed parameterized components. Communications are crucial to their overall functionality and performance. Formal verification methods have been intensively applied to processing elements, e.g., microprocessors. Very little work has been done with respect to communication modules. We present the formal specification of a packet switched NoC and its proven refinement. At the specification level, routing decisions are computed *at once* before packets get injected in the network. In the implementation, routing decisions are distributed over each individual node. We prove that the implementation behaves according to its specification for a 2D-mesh NoC. All models and proofs have been checked using the ACL2 theorem proving system. To the best of our knowledge, this work constitutes the first cross-layer verification of on-chip communication networks.**

I. Introduction

Formal verification often consists in showing that for every execution of an implementation there exists an execution of its abstract specification with the same visible effects. This approach has been successfully applied to processing elements (e.g. microprocessors [7], [8]). Multi-Processor Systems-on-Chip designs offer increased performance by combining several processing and memory cores on a single die. The interconnect is becoming crucial to the overall functionality of an MPSoC [13]. When the number of interconnected units grows, bus performances decrease. Networks-on-Chips (NoCs) [2] is a solution that could meet future system performance.

Regarding buses, the recent work of Böhm and Melham is the only effort trying to fill the gap between abstract specifications and low level implementations [3]. Previous efforts concentrate on proving properties on low-level implementations using model-checking [11] or combination of model-checking with theorem proving [1]. Gebremichael *et al.* [5] provide a parametric analysis of part of the AEthereal NoC [6]. All these works considers *implementations* only. Regarding specifications, Schmaltz *et al.* [12], [4] propose a generic network model, named GeNoC. We present models that are variations of the GeNoC model. Each model is at a different abstraction level. Our contribution is a formal relation between instances of these models.

Our goal is to provide a methodology to support the abstract specification of NoCs and the proof that implementations conform to it. In this paper, we present an initial effort towards this goal. We present the formal specification of a packet switched NoC. At the specification level, routing decisions are computed *at once* before packets get injected in the network. In the implementation, routing decisions are distributed over each individual node, i.e., hop-by-hop. Details of the implementation model are available [14]. The original contribution of this paper consists of the definition of the specification model and the proof that the implementation behaves according to its specification for a 2D-mesh network based on the HERMES NoC [10]. All models and proofs have been checked using the ACL2 theorem proving system [9] and are available on the web[1]. To the best of our knowledge, this paper presents the first cross-layer verification of a NoC.

II. A NoC Example: HERMES

HERMES [10] is based on a 2D mesh architecture (Fig. 1). Each node is made of an IP core and a switch. Each switch has five bi-directional ports: East, West, North, South connecting to the neighbor switches, and Local to the IP core.

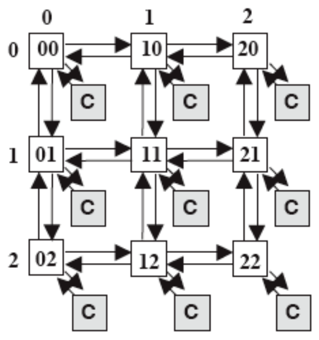

Fig. 1. Mesh architecture [10]

The routing policy is based on a deterministic, minimal algorithm: the *XY routing algorithm*. Each packet is routed on one dimension at a time. It travels first along the X axis until the first coordinate of the destination is reached, and then travels along the Y axis. This algorithm is recalled in Fig. 2.

Our HERMES instance uses *packet switching* (store-and-forward). A packet constitutes the fixed size basic unit, which travels in the network. A packet contains a *header* with routing information and a payload with data. A packet is sent autonomously through the network.

Each port of the switch has an input buffer queue. A Round Robin priority policy is used to access the output ports. If the requested port is busy, packets are blocked and the request signal remains active. The transmission between two different nodes is ruled by an handshake protocol.

[1] www.cs.ru.nl/∼julien/Julien_at_Nijmegen/FMCAD09.html

```
XYRouting(from,to) :
if from=to     /* destination reached */
then take the Local port
else
  if Xfrom != Xto
  then     /* change X */
       if Xfrom < Xto
       then take port East
       else take port West
  else     /* change Y */
       if Yfrom < Yto
       then take port South
       else take port North
```

Fig. 2. XY routing algorithm

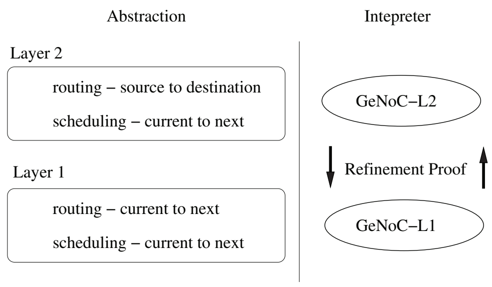
Abstraction
Intepreter
Layer 2
routing – source to destination
scheduling – current to next
GeNoC–L2
Refinement Proof
Layer 1
routing – current to next
scheduling – current to next
GeNoC–L1

Fig. 3. Verification Approach

III. Verification Approach

A. NoC Abstraction, Interpreter, and Property

Our method consists of a NoC model, two abstraction levels for that model, and a NoC interpreter for each level. Both interpreters are simulators for networks composed of identical nodes following a generic router model. They are proven equivalent (Section VII). The generic model has a few design-dependent functions which constitute the user-input. These functions are the Link Layer protocol (e.g., handshake), the routing logic (e.g., XY), and the scheduling policy (e.g., packet switching). Together with the router model, they are described below. The main difference between the interpreters lies in the routing decisions. The specification (Layer 2) supports *source* routing, where routes are computed before sending packets. The implementation supports *distributed* routing where each node computes the next step in a route.

B. Network Model

We assume a generic architecture composed of an arbitrary – but finite – number of nodes and a finite number of connections between any two nodes. Each node is uniquely identified by its position. A node includes a local memory and a router. A router is defined by a set of ports and four functions: input and output units, routing control, and flow control (see Fig. 4). All nodes are identical.

The main elements of a port are the data and control signals, and internal buffers (Fig. 4). Formally, a port is a tuple $\langle addr, stat, data, buff \rangle$, where $addr$ is a unique address, $stat$ stores the values of the control signals and other state

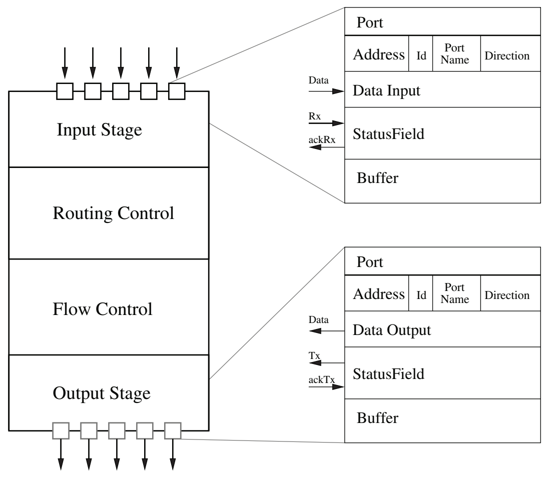

Fig. 4. A router and its ports

components, $data$ denotes the values of the data signals, and $buff$ represents the value of the buffers.

An address is a tuple $\langle coor, pid, dir \rangle$, where $coor$ is the unique identifier of the node the port belongs to, pid the name of the port (e.g., *west*, *south*), and dir the direction, i.e., 'i' for an input port or 'o' for an output port. The topology is a list where each element is a pair of port addresses (p_i, p_j), which means that port p_i is connected to port p_j. A node is defined as the set of ports, where the address of each port p is the same. These ports define the state of the node. The set of all ports of a network defines the state of the network.

Functions `ProcessInputs` and `ProcessOutputs` define the low level protocols which use the control signals to transfer the content of the data signals to the internal buffers in case of an input port, or to transfer the content of the buffers to the data signals in case of an output port.

Function `RouteControl` applies the routing logic to one or more ports of a node. It returns a list of *routed ports*, i.e., ports together with routing information. The only design-dependent function is function `routing-logic` which implements the routing algorithm.

Function `FlowControl` implements the switching technique, e.g., packet, circuit, or wormhole. In case of conflict, this function also resolves priorities. This function extracts from the routed ports the packets that are ready to be transmitted. The only design-dependent function is function `switch-ports` which effectively schedules packets. Those scheduled packets are moved to the output ports computed by the routing control function.

All these functions form function *router* (Fig. 5), which updates a node state. Note that a node is equipped with a memory which is available to each port and each function. Argument `nstmem` represents that memory. To simplify the presentation, we assume that such a memory element is given as input argument of any function that accesses it. This argument is not explicitly mentioned any further.

```
router(nst,nstmem) :
let (nst nstmem) be
    RouteControl(ProcessInputs(nst), nstmem)
    in
    let (nst nstmem) be
        FlowControl(nst, nstmem)
        in
        return ProcessOutputs(nst), nstmem
```

Fig. 5. Function `router`

IV. Network Model Instance – User Input

For the HERMES 2D-mesh, an address is defined by the xy-coordinate of the node, $pid \in \{n, s, e, w\}$, and $dir \in \{i, o\}$. A topology element identifies bi-directional links between two nodes, for instance $(\langle 0\ 0, e\rangle, \langle 1\ 0, w\rangle)$.

The input and output units implement an handshake protocol. A node can request the transmission using signal `Tx` connected to signal `Rx` of the receiver node. The latter can deny or grant the access using signal `AckRx` connected to signal `AckTx` of the sender.

Function `routing-logic` is defined to implement the routing algorithm described in Fig. 2.

Fig. 6 shows the instantiation of function `switch-ports` for packet switching, named `pkt-switch-ports`. It takes as arguments the list of the output ports (`outports`); an input port (`from`), the content of which has been routed; and the state of the node (`nst`). Function `pkt-switch-ports` finds the output to which the input port must be connected, and checks whether this port can accept the packet. Each node has a one-place output buffer for each output port. A port accepts a packet if its buffer is empty. If such a port exists, function `switchBuffer` transfers the content of the input port to the output port, i.e., loads the output port and clears the input port.

```
pkt-switch-port(outports, from, nst)
let to be outports[0]
    in
    case
       outports = null: return nst

       status-route(port-status(from))
       and not (port-bufferFull(to)):
       return switchBuffer(nst,from,to)

       default:
       pkt-switch-port(outports[1..]), from,nst)
```

Fig. 6. Function `pkt-switch-port`

V. Specification – GeNoC-L2

Function *GeNoC*-L2 (Fig. 7) is the core function of our interpreter for the specification layer. Input argument `simL` defines the length of the simulation. It takes as additional arguments the set of packets to be sent (`m`), the current state of the network (`ntkst`), an accumulator of packets that have reached their destination (`arr`, initially empty), the current simulation step (`z`, initially 0), and the topology (`topo`). It returns the list of arrived packets, the list of delayed packets, and the state of the network at the end of the simulation.

```
GeNoC-L2(m, ntkst, arr, z, topo, simL) :
if simL = 0 return arr,m,ntkst
else
 let (dep, del) be
     depart-L2(ntkst, m, z)
     in
     let newntkst be
         step-ntk-L2(dep, topo)
         in
         GeNoC-L2(del,newntkst, add(z,arr),
                  z + 1,topo, simL-1)
```

Fig. 7. Function `GeNoC-L2`

Function `depart-L2` controls packet injection. According to a user-defined criterion, it determines which packets can be in the network. At the specification level, this function uses *source routing* and appends to a packet its route *from its source to its destination*. It inserts this extended packet in the local input port of its source node. It returns a new state (`dep`) and a list of delayed packets (`del`). Function `step-ntk-L2` (see below) actually performs one simulation step. It updates the global network state. Those packets that are at their destination are extracted from this new state and appended to accumulator *arrived*. The next recursive call processes the delayed packets, the new network state, and time is incremented by 1.

```
step-ntk1-L2(ntslist, ntkst):
if ntslist = null return ntkst
else
let newnst be ProcessInputs-L2(ntslist) in
    let (newnstlist,newntkmem1) be
        RouteControl-L2(newnstlist,ntkmem) in
        let (newnstlist, newntkmem) be
            FlowControl-L2(newnstlist, newntkmem1)
            in
            return ProcessOutputs-L2(newnstlist),
                   newntkmem

step-ntk-L2(ntkst, topo):
let newntkst be
    step-ntk1-L2 (ports-nodelist(ntkst), ntkst)
    in
    updateNeighbours-L2(newntkst,topo)
```

Fig. 8. Function `step-ntk-L2`

At the specification level, every element of our router is applied sequentially over all nodes. Function `step-ntk1-L2` (Fig. 8) takes as arguments a list of nodes to be processed (`ntslist`) and the current network state (`ntkst`). It updates the network state. Function `ProcessInputs-L2` applies the input stage to all nodes. As routes have been computed at injection time, the RouteControl module simply ”reads” the next hop as the first element of the route. Function `RouteControl-L2` performs this ”routing” decision at all nodes. After that, function `FlowControl-L2` applies the packet switching policy to all nodes. Finally, function `ProcessOutputs-L2` applies the output stage at all nodes.

Function `step-ntk` extracts the node structures from the list of ports (function `ports-nodelist`), and calls `step-ntk1-L2`. Function `updateNeighbours-L2` simulates the transfer of data from output data signals to input data signals. This function removes the first element of routes.

VI. Implementation – GeNoC-L1

Function *GeNoC*-L1 is the core function of our interpreter for the implementation layer. It takes and returns the same arguments as *GeNoC*-L2. Its definition is obtained by replacing every occurrence of "L2" with "L1" in Fig. 7. It only injects a packet without appending any additional information.

```
step-ntk1-L1(ntslist, ntkst):
if ntslist = null return ntkst else
let newnst be router(ntslist[0]) in
    let newntkst be
        step-ntk1-L1(ntslist[1..], ntkst) in
        return ports-update(newntkst,newnst)
step-ntk-L1(ntkst, topo):
let newntkst be
    step-ntk1-L1 (ports-nodelist(ntkst), ntkst) in
    updateNeighbours-L1(newntkst,topo)
```

Fig. 9. Function `step-ntk-L1`

Function `step-ntk-L1` (Fig. 9) is based on recursive function `step-ntk1-L1`. The latter takes as arguments a list of nodes to be processed (`ntslist`) and the current network state (`ntkst`). It updates the network state. For each node, it applies function `router`. Function `ports-update` effectively updates the state of the nodes. Finally, function `step-ntk-L1` extracts the node structures from the list of ports (function `ports-nodelist`), and calls `step-ntk1-L1`. Function `updateNeighbours-L1` simulates the transfer of data from output to input signals.

VII. Equivalence Proof

The theorem connecting the two models is shown in Fig. 10. Function `transform` simply removes all routes from extended packets. This theorem states that after the application of `transform` the lists of arrived packets, the lists of packets still en route in the network, and the final network state produced by *GeNoC*-L2 equals those produced by *GeNoC*-L1. The proof in itself is nothing deep. The two interpreters manipulate the same functions. The only difference is in the ordering of these function calls. The difficulties lie in getting the right model definitions and the details of the formal proofs.

Our proof depends on three axioms about the topology and the state generated from it. They basically state that the initial network state is well-formed, e.g., it agrees with the topology.

VIII. Conclusion and Future Work

We presented the first effort in building a verification methodology of NoCs. We defined two abstractions layers and proved their equivalence. The source routing specification is correctly refined into a distributed routing implementation. A large part of our model and the proof is design-independent. Our plan is to extract the generic part of our proof and

```
Theorem:
let (arr-l1, m-L1, ntkst-L1) be
    GeNoC-L1(m, ntkst, arr, z, topo, simL) in
    let (arr-l2, m-L2, ntkst-L2) be
        GeNoC-L2(m, ntkst, arr, z, topo, simL) in
        transform(arr-L2) = arr-L1 and
        m-L2 = m-L1 and
        transform(ntkst-L1) = ntkst-L2
```

Fig. 10. Equivalence theorem

obtain a general verification method. We also are working on extending GeNoC-L2 to support global and application-independent properties like functional correctness or deadlock avoidance [15]. We are convinced that the structure of our implementation is similar to the actual structure of RTL designs. One has now to relate our algorithm to the RTL. We plan to investigate the generation RTL code from our models.

Acknowledgments

This research is supported by NWO/EW project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811.

References

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

[2] L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.

[3] P. Böhm and T. Melham. A refinement approach to design and verification of on-chip communication protocols. In *Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE Computer Society, 2008.

[4] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A formal approach to the verification of networks on chip. *EURASIP Journal on Embedded Systems*, 2009(Article ID 548324):14 pages, 2009.

[5] B. Gebremichael, F. Vaandrager, and M. Zhang. Formal Models of Guaranteed and Best-Effort Services. Tech. Rep. Institute for Computing and Information Sciences, Radboud University Nijmegen, March 2005.

[6] K. Goossens, J. Dielissen, and A. Rădulescu. Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, September-October 2005.

[7] W. A. Hunt. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[8] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. *Computer-Aided Verification CAV*, LNCS 818, pages 68–80, Standford, California, USA, 1994. Springer.

[9] M. Kaufmann, P. Manolios, and J Strother Moore. *ACL2 Computer Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[10] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration*, 38(1):69–93, 2004.

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

[12] J. Schmaltz and D. Borrione. A functional formalization of on chip communications. *Formal Aspects of Computing*, 20(3):239–348, 2008.

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

[14] T. van den Broek and J. Schmaltz. A generic implementation model for the verification of networks-on-chips. In *8th Intl. Workshop on the ACL2 Theorem Prover and Its Application*, pages 130–134, Northeastern Univ., Boston MA, USA, 2009. ACM.

[15] F. Verbeek and J. Schmaltz. Formal validation of deadlock prevention in networks-on-chips. In *8th Intl. Workshop on the ACL2 Theorem Prover and Its Application*, pages 135–145, Northeastern Univ., Boston MA, USA, May 11–12 2009. ACM.