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 AEtheral 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 web¹. 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.

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.

Regarding the switch, which is the only port that can block the transmission of packets, we use a Round Robin priority policy.

FIGURE 1. Mesh Architecture [10]

¹www.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

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 \((addr, stat, data, buff)\), where \(addr\) is a unique address, \(stat\) stores the values of the control signals and other state components, \(data\) denotes the values of the data signals, and \(buff\) represents the value of the buffers.

An address is a tuple \((coor, pid, dir)\), 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 \texttt{ProcessInputs} and \texttt{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 \texttt{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 \texttt{routing-logic} which implements the routing algorithm.

Function \texttt{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 \texttt{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 \texttt{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 \texttt{nstrmem} 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.
A topology element identifies bi-directional links between two nodes, for instance \(((0,0),(1,0),(0,1))\).

The input and output units implement a 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 \(AckRx\).

Function \(\text{routing-logic}\) is defined to implement the routing algorithm described in Fig. 2.

Fig. 6 shows the instantiation of function \(\text{switch-ports}\) for packet switching, named \(\text{pkt-switch-ports}\). It takes as arguments the list of the output ports (\(\text{outports}\)); an input port (\(\text{from}\)), the content of which has been routed; and the state of the node (\(\text{nst}\)). Function \(\text{pkt-switch-ports}\) 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 \(\text{switchBuffer}\) transfers the content of the input port to the output port, i.e., loads the output port and clears the input port.

Function \(\text{depart-L2}\) 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 (\(\text{dep}\)) and a list of delayed packets (\(\text{del}\)). Function \(\text{step-ntk-L2}\) (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 \(\text{arr}\). The next recursive call processes the delayed packets, the new network state, and time is incremented by 1.
Function `step-ntk` extracts the node structures from the list of ports (function `ports-nodelist`), and calls `step-ntkl-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-ntkl-L1(ntslist, ntkst):
    if ntslist = null return ntkst else
    let newntkst be router(ntslist[0]) in
    step-ntkl-L1(ntslist[1..], ntkst) in
    return ports-update(newntkst,newntkst)
step-ntk-L1(ntkst, topo):
    let newntkst be
        step-ntkl-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-ntkl-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-ntkl-L1`. Function `updateNeighbours-L1` simulates the transfer of data from output to input signals.

VII. EQUVALENCE PROOF

The theorem connecting the two models is shown in Fig. 10. Function `transform` simply removes all routes from extracted 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 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