A Generic Implementation Model for the Formal Verification of Networks-on-Chips

Tom van den Broek and Julien Schmaltz[*]
Radboud University Nijmegen
Institute of Computing and Information Sciences
6500 GL Nijmegen, The Netherlands
tombroek@science.ru.nl,julien@cs.ru.nl

ABSTRACT

Formal verification often means the proof of a formal relation between abstract specification models and concrete implementation models. For microprocessor designs, commutative diagrams derived from these models and relations have been very successful. In the context of communication modules, no such diagram exists. The generic network-on-chip model (GeNoC) has been recently proposed as a generic *specification* model to validate high-level descriptions of networks-on-chips. We report on work in progress towards the definition of a generic verification diagram based on GeNoC. We present a generic model for *implementations.* Following the GeNoC approach, our new model is *generic* in the sense that it characterizes a large family of designs and that the validation of a concrete implementation consists in proving it a valid instance of the generic model. In the paper, we detail the implementation of packet and circuit switching techniques. We report on other instances which support the generic character of our model.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*verification*

General Terms

algorithms, verification

Keywords

formal verification, networks-on-chips

1. INTRODUCTION

Formal verification often involves a specification, an implementation, and the proof that for all executions of the implementation there exists an execution of the specification that has the same effect. In the context of microprocessors, this consists in relating the sequential execution of the Instruction Set Architecture to the pipelined execution of the Register Transfer Level design [6, 5, 7]. This approach has been very successful. Simple industrial designs can be fully verified, and the verification effort is largely automatic [2]. Today, systems-on-chips are multi-processors and their functional correctness largely depends on the correctness of the communication architecture [4]. Until recently, most of the verification effort was concerned with processing elements. The sparse literature devoted to communication modules proposes ad hoc solutions which apply to particular designs described at a low level of abstraction. In contrast, the *generic network-on-chip* (GeNoC) model offers a general specification and validation environment for high-level descriptions of NoCs [9, 1]. GeNoC has been implemented in ACL2 and applied to several case-studies.

Our goal is to build a layered verification environment for NoCs. For specification layers, GeNoC offers abstractions which simplify the formal validation. For instance, entire routes from source to destination nodes are computed as one function call, and *after that* the scheduling policy is applied to determine the set of communications which can make a hop. We sketch the main elements of a new model which breaks this abstraction, i.e., routes, as well as scheduling, are computed step-by-step. This new model, its formalization in ACL2, and its application to several concrete instances constitute the original contribution of this paper. The proof of a relation between this new model and GeNoC is still future work.

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

2. GENOC IMPLEMENTATION MODEL

2.1 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. 1). All nodes are identical.

Ports, topology and state. The main elements of a port are the data and control signals, and internal buffers (Fig. 1). 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 components of a port, $data$ denotes the values of the data signals, and $buff$ represents the value of the buffers associated with the port.

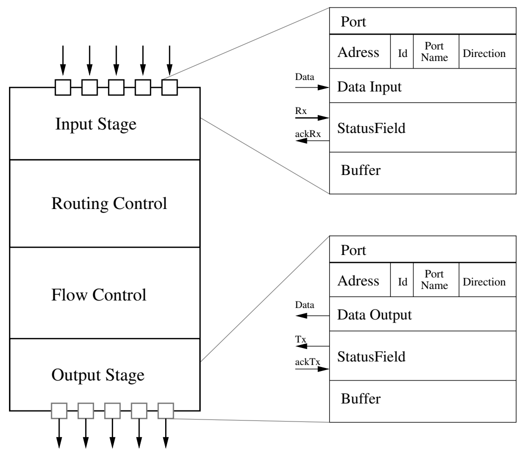

Figure 1: A router and its ports

An address is a tuple $\langle coor, pid, dir \rangle$, where *coor* is the unique identifier of the node the port belongs to, *pid* is the name of the port (e.g., *west*, *south*), and *dir* is 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.

Example 1. *Assume a 2D-mesh. Each node has five ports: one for each direction (north, east, west, and south), and one for the local core. An address is then 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* $(\langle 0\ 0, e\rangle, \langle 1\ 0, w\rangle)$. *Examples of ports are detailed in Section 3.*

Input and output units. These two functions 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.

Example 2. *A handshake protocol is an instance of the input and output units. 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.*

Routing control. This function 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 function that needs to be instantiated is function `routing-logic` which implements the routing algorithm.

Example 3. *Dimension order routing [3] is a popular routing algorithm. In particular, the XY routing algorithm for a 2D-mesh. Messages travel along the X axis completely, and then along the Y axis. The core control logic of such implementations – like the Hermes NoC [8] – computes the next hop as a simple function of the current position and the destination. This functionality is exactly the purpose of instances of function* `routing-logic`.

Flow control. This function implements the switching technique, e.g., packet, circuit, or wormhole. In case of conflict, this function also resolves priorities. Function *flowcontrol* extracts from the routed ports the messages that are ready to be transmitted. The core function that needs to be instantiated is function *switch-ports* which effectively schedules messages. Those scheduled messages are moved to the output ports computed by the routing control function. We detail in Section 3 two instances of this function.

Global definition. All these functions form function *router* (Listing 1), 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.

```
(defun router (nst nstmem)
  (mv-let
    (nst nstmem)
    (RouteControl (ProcessInputs nst)
                  nstmem)
    (mv-let (nst nstmem)
            (Flowcontrol nst nstmem)
            (mv (ProcessOutputs nst)
                nstmem))))
```

Listing 1: Function *router*

2.2 Network interpreter

Function $GeNoC_t$ (Listing 2) is the core of our interpreter. It works as a simulator which applies function `router` to each node. Each recursive call defines a simulation step. Input argument `simL` defines the length of the simulation. Function $GeNoC_t$ takes as additional arguments the set of messages to be sent (`m`), the current state of the network (`ntkst`), an accumulator of messages 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 messages, the list of delayed messages, and the state of the network at the end of the simulation.

```
(defun genoc_t (m ntkst arr z topo simL)
  (if (zp simL)
    (mv arr m ntkst)
  (mv-let
    (dep del) ;; dep = new value of ntkst
    (depart ntkst m z)
    (let
      ((newntkst (step-ntk dep topo))
      (genoc_t
        del newntkst
        (append
          (list (list 'TIME z
                  (arrive newntkst)))
           arr)
        (1+ z) topo (1- simL)))))))
```

Listing 2: The GeNoC function

Function `depart` controls message injection. According to a user-defined criterion, it determines which messages can be in the network (line 5). These messages have either already left their source or `depart` inserts them in the local input port of their source node. Function `depart` returns a list of updated nodes (`dep`) and a list of delayed messages (`del`). Function `step-ntk` (see below) applies function `router` to each node (line 7). This produces a list of updated nodes. Those messages that are at their destination are extracted from this new state and appended to accumulator *arrived* (lines 10 to 13). The next recursive call processes the delayed messages, the updated nodes, and time is incremented by 1.

Function `step-ntk` (Listing 3) is based on recursive function `step-ntk1`. 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` (line 4). Function `ports-update` effectively updates the state of the nodes (line 7). Finally, function `step-ntk` extracts the node structures from the list of ports (function `ports-nodelist`), and calls `step-ntk1`. Function `updateNeighbours` simulates the transfer of data from output data signals to input data signals.

```
(defun step-ntk1 (ntslist ntkst)                    0
  (if (endp ntslist)
    ntkst
    (let*
     ((newnst  (router (car ntslist)))
      (newntkst                                     5
        (step-ntk1 (cdr ntslist) ntkst)))
     (ports-update newntkst newnst))))

(defun step-ntk (ntkst topology)
  (let                                              10
   ((newntkst
    (step-ntk1 (ports-nodelist nktst nil)
               ntkst)))
   (updateNeighbours newntkst topology)))
```

Listing 3: Functions `step-ntk` and `step-ntk1`

3. APPLICATION TO FLOW CONTROL

Listing 4 shows the definition of function `FlowControl`. Function `switch-nst` takes a list of input ports extracted from the network state by function `ports-inputports`. At each input port it applies the scheduling policy of the network. This policy is represented by function `switch-port`, which is the only function to be implementation dependent. In the next two sections, we give it two different definitions.

```
(defun FlowControl (nst)
  (switch-nst (ports-inputports nst) nst))
```

Listing 4: Function `FlowControl`

3.1 Packet switching

Packet switching encodes messages in *packets*. A packet constitutes the fixed size basic unit, which travels in the network. A packet contains a *header* which contains routing information and data. A packet is sent autonomously through the network. After a packet has arrived at an input port and has been routed by the routing control, it is sent to the correct output port if the latter can accept the packet, either because it has space to store it, or because the input port of the next node has available space. Otherwise, the packet is stored in the input or output port of its current position. We have implemented the first solution. In our example, each node has a one-place output buffer for each output port. A port accepts a packet if its buffer is empty.

Listing 5 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 (lines 4 to 6), and checks whether this port can accept the message (line 7). If such a port exists, function `switch-Buffer` transfers the content of the input port to the output port, i.e., loads the output port and clears the input port.

```
(defun pkt-switch-port (outports from nst)   0
  (let ((to (car outports)))
    (cond
      ((endp outports) nst)
      ((and (equal (port-portname to )
            (status-route                    5
                 (port-status from)))
            (not (port-bufferFull to)))
        (switchBuffer nst from to))
      (t (pkt-switch-port
         (cdr outports)  from nst)))))       10
```

Listing 5: Flow control: packet switching

3.2 Circuit switching

Circuit switching is a scheduling technique where before any data can be sent a *circuit* must be allocated from the origin to the destination. As long as a circuit is allocated to a source and a destination port, the ports of this circuit cannot be used for another circuit.

Between a source node s and a destination node d, a circuit is created in two steps. First, the source port sends a `request` packet to the destination port. When traveling towards this destination, the request packet temporarily books each intermediate port by setting the status of these ports to `requested`. If the request reaches the destination, an acknowledgment is sent back following the same path. The acknowledgment confirms the booking of the ports by setting their status to `booked`. A circuit is established only if the request reaches its destination. Once a circuit is booked, the source port sends an arbitrary (finite) number of data packets. Then, it sends a `torn-down` packet which clears the circuit. We have implemented a variation of this technique, where the request packet contains the number of data packets that will use the circuit. Therefore, each intermediate port knows how many packets must be transmitted, and clears the circuit when the last packet has been forwarded. No `torn-dow` packet is required.

Part of the instantiation of function `switch-port` for circuit scheduling (`ct-switch-port`) is given in Listing 6. It only shows how a circuit acknowledgment packet is emitted (lines 21 to 32) and transmitted back to its source port (lines 6 to 17). An acknowledgment packet is created if the request packet has reached an output local port with available space (lines 21 to 24) and this local port has not been booked yet (line 25). Function `sendAck` places an `ack` on the output ports given as its second argument. Its first argument is the value of the node state after moving the request from the input port to the local output port (function `switch-buffer`, line 28), and storing the circuit in the port (function `update-circuit`, lines 27 to 31).

This "booking" is continued all the way back to the source node. At each intermediate node, if a port receives an `ack` (line 6), it has available space and a request has been previously stored (lines 7 to 12)), function `update-circuit` sets the port status to `booked`, stores the circuit information, and forwards the `ack` to the next node.

```
(defun switch-port (outports from nst)              0
 (let ((to (car outports)))
  (cond
    ((endp outports) nst)
    ...
    ((and (equal (port-buffer from)                 5
                 '(ack))
          (endp (port-buffer to))
          (equal (port-dir to) 'out)
          (equal (port-circuitState to)
                 'request)                          10
          (equal (port-circuitId to)
                 (port-portname from)))
     (updateCircuit
      (switchBuffer nst from to)
      (port-portname to)                            15
      (port-portname from)
      'booked))
    ...
    ((and (equal (port-portname to)
                 (status-route                      20
                    (port-status from)))
          (endp (port-buffer to))
          (equal (port-dir to) 'out)
          (equal (port-portname to) 'loc)
          (not (port-circuitState to)))             25
     (sendack
      (updateCircuit
        (switchBuffer nst from to)
        (port-portname to)
        (port-portname from)                        30
        'booked)
      (port-portname from)))
    ...)))
```

Listing 6: Flow control: circuit switching

4. CONCLUSION AND FUTURE WORK

We have presented an initial formalization of a generic model for NoCs implementations. We showed details on the instantiation of its scheduling function for the packet and switching techniques. The generic character of our model is supported by other instances, summarized in Table 1.

Generic comp.	Instances	lines	functions
interpreter		231	12
types		314	41
router		170	8
routing control	xy	150	11
	spidergon	126	10
flow control	packet-switching	70	4
	wormhole	97	9
	circuit-switching	125	10
departure	injection time	29	1
input and outputs	handshake	129	15
Total		1441	121

Table 1: Instances of our implementation model

A verification diagram for NoCs can now be obtained from a formal relation between the GeNoC specification model and the GeNoC implementation model presented in this paper. Providing this relation is precisely the subject of our current investigations.

5. REFERENCES

[1] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. Executable formal specification and validation of NoC communication infrastructures. In *Proceedings of the 21st annual symposium on Integrated circuits and system design (SBCCI'08)*, pages 176–181, Gramado, Brazil, September 1–4 2008. ACM.

[2] W. Büttner. Is Formal Verification Bound to Remain a Junior Partner of Simulation? In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, 2005. Invited Speaker.

[3] W. Dally and C. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[4] G. de Micheli and L. Benini. *Networks on Chips.* Elsevier, 2006.

[5] W. Hunt. Mechanical mathematical methods for microprocessor verification. In *CAV*, pages 523–533, 2004.

[6] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 68–80, Standford, California, USA, 1994. Springer-Verlag.

[7] P. Manolios and S. Srinivasan. A framework for verifying bit-level pipelined machines based on automated dedution and decision procedures. *Journal of Automated Reasoning*, 37(1–2):93–116, 2006.

[8] 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.

[9] J. Schmaltz and D. Borrione. A functional

formalization of on chip communications. *Formal Aspects of Computing*, 20(3):239–348, 2008.