Formal Specification and Validation of Minimal Routing Algorithms for the 2D Mesh

Julien Schmaltz
Radboud University
Institute of Computing and Information Sciences
6500 GL Nijmegen, The Netherlands
julien@cs.ru.nl

ABSTRACT

This paper presents the formalization of a family of routing algorithms in the ACL2 logic. This work is based on *GeNoC*, a *generic model* of networks on a chip (NoCs). We review the principles of this approach. We detail the specification and the validation of dimension-order routing for the 2D mesh. We consider deterministic and adaptive algorithms.

Categories and Subject Descriptors

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

General Terms

algorithms, verification

Keywords

routing algorithms, networks on chip, formal methods

1. INTRODUCTION

Chip business is highly competitive and time-to-market is ever shrinking. A three month delay induces a loss of a fourth of the expected income [3]. To meet this demand, *systems on a chip* (SoCs) design relies on a *platform* based approach: a new SoC is built by assembling pre-designed parameterized modules. Hence, the specification and the validation of the interconnect becomes crucial [13].

Networks on a chip (NoCs) constitute a recent paradigm, which could meet future SoC's performance requirements [1]. Little work has been done with respect to their formal verification. One notable exception is the formal analysis of the Æthereal protocol [6] of Philips in the PVS logic [8] by Gebremichael *et al.* [5]. This work was performed on a very specific design.

In contrast, this paper is based on a more general approach [11, 2], based on a functional representation of the communications. It relies on a generic network model (named *GeNoC*), which formalizes the interactions between three key components: interfaces, routing and scheduling. To abstract from any particular design, no concrete definition is given to these components. A component is only defined using constraints, or *proof obligations*. Those imply the global correctness of *GeNoC*. Consequently, the validation of a particular design is reduced to the verification of these constraints.

GeNoC has been used to validate different architectures, *e.g.* wormhole routing [2] (see [10] for more details). Two dimensional grids constitute popular architectures. *GeNoC* has been applied to deterministic networks based on this structure. In this paper, we show how to extend these results to *adaptive* routing algorithms.

In the next section, we briefly summarize the principles of *GeNoC*. We give a precise description of the proof obligations associated with the routing module of *GeNoC*. Section 3 formalizes basic notions for deterministic networks. Based on these concepts, we present the formalization and the validation of an adaptive routing algorithm in Section 4. Section 5 concludes the paper and discusses future work.

2. A GENERIC NETWORK ON CHIP

2.1 A General Model of Communications

GeNoC considers the general communication model of Figure 1. An arbitrary, but finite, number of *nodes* are connected to some communication architecture. The latter represents the interconnection structure, *e.g.* bus or network. It comprises topologies, routing algorithms and scheduling policies. Our model makes no assumption on these components. As proposed by Rowson and Sangiovanni-Vincentelli [9], each node is separated into an *application* and an *interface*. The latter is connected to the communication architecture. Interfaces allow applications to communicate using protocols. Any interface-application pair matches the layers of the OSI model. Interfaces generally refer to layers 1 to 4; applications to layers 4 to 7. Layer 4 is a boundary and can be part of either interfaces or applications. To distinguish between interface-application and interface-interface communications, an interface and an application communicate using *messages*; two interfaces communicate using *frames*.

Applications represent the computational and functional aspects of nodes. They are either active or passive. Typically, active applications are processors and passive applications memories. We consider that each node contains one passive and one active application, *i.e.* each node is capable of

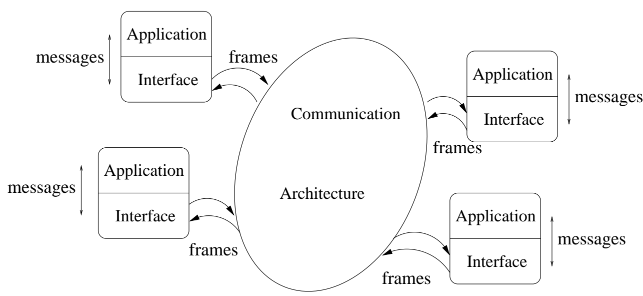

Figure 1: Communication Model

sending and receiving frames. As we want a general model, applications are not considered *explicitly*: passive applications are not actually modeled, and active applications are reduced to the list of their pending communication operations. We focus on communications between distinct nodes. We suppose that in every communication, the destination node is different from the source node.

2.2 Overview of *GeNoC*

Function *GeNoC* represents the transfer of messages from their source to their destination. Its main argument is the list of messages emitted at source nodes. It returns the list of the results received at destination nodes.

Interfaces. Function *send* represents the encapsulation of a message into a frame. Function *recv* represents the decoding of this frame to recover the emitted message. The main constraint associated with these functions expresses that a receiver should be able to extract the encoded information, *i.e.* the composition of function *recv* with function *send* ($recv \circ send$) is the identity function.

Routing Algorithm. The routing algorithm is represented by the successive application of unitary moves (routing hops). For each pair made of a source node s and a destination node d, the routing function computes *all* possible routes available between s and d. The main constraint associated with the routing function expresses that each route from s to d effectively starts in s and uses only existing nodes to end in d.

Switching Technique. The scheduling policy participates in the management of conflicts that appear on the network. It defines the set of communications that can be performed *at the same time*. Formally, these commutations satisfy an *invariant*. Scheduling a communication, *i.e.* adding it to the current set of authorized communications, must preserve the invariant, for all times and in any admissible state of the network. The invariant is specific to the scheduling policy. In our formalization of the scheduling policy, the existence of this invariant is assumed but not explicitly represented. From a list of requested communications, the scheduling function extracts a sub-list that satisfies the invariant. The rest makes up the list of delayed communications.

Function *GeNoC*. Function *GeNoC* is pictured in Fig. 2. It takes as arguments the list of requested communications and the characteristics of the network. It produces two lists as results: the messages received by the destination of successful communications and the aborted communications.

The main input of *GeNoC* is a list $\mathcal{T}$ of *transactions* of the form $t = (id\ A\ msg_t\ B)$. Transaction t represents the intention of application A to send a message msg_t to application B. A is the *origin* and B the *destination*. Both A and B are members of the set of nodes, *NodeSet*. Each transaction is uniquely identified by a natural *id*.

Briefly, function *GeNoC* works as follows. For every message in the initial list of transactions, it computes the corresponding frame using *send*. Each frame together with its *id*, *origin* and *destination* constitutes a *missive*. Then, *GeNoC* computes the routes of the missives and schedules them using functions *Routing* and *Scheduling*. To keep our model general, function *Routing* computes a list of routes for every missive. If the routing algorithm is deterministic, this list has only one element. Once routes are computed, a *travel* denotes the list composed of a frame, its *id* and its list of routes. The results of the scheduled travels are computed by calling *recv*. The *delayed* travels are converted back to missives and constitute the argument of a recursive call to *GeNoC*. To make sure that this function terminates, we associate to every node a *finite* number of attempts. At every recursive call of *GeNoC*, every node with a pending transaction will consume one attempt. Function *GeNoC* halts if every attempt has been consumed. The first output list $\mathcal{R}$ contains the results of the completed transactions. Every result r is of the form $(id\ B\ msg_r)$ and represents the reception of a message msg_r by its final destination B. Transactions may not run to completion (*e.g.* due to network contention). The second output list of *GeNoC* is named *Aborted* and contains the canceled transactions.

Function *GeNoC* is considered correct if every non aborted transaction $t = (id\ A\ msg\ B)$ is completed in such a way that B effectively receives *msg*. Formally, we prove that for every final result r, there is a unique initial transaction t such that t has the same *id* and *msg* as r.

$$\forall rst \in \mathcal{R}, \exists! t \in \mathcal{T}, \begin{cases} & Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge & Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge & Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{cases} \quad (1)$$

This formula is proved a theorem using the proof obligations associated with each component. As shown in this formula, the pencil and paper formulation of *GeNoC* relies on the quantification over variables, as well as functions. We have developed a systematic approach to translate this mathematical model to the ACL2 logic [12]. In the remainder of this section, we only detail aspects relative to routing algorithms.

2.3 Formalization of Routing Algorithms

2.3.1 Nodes and Parameters

Nodes are defined on an arbitrary domain, *GenNodeSet*. A list of elements of that domain is recognized by predicate *NodeSetp*, which is a constrained function. The set of nodes of a particular network is noted *NodeSet*. It is

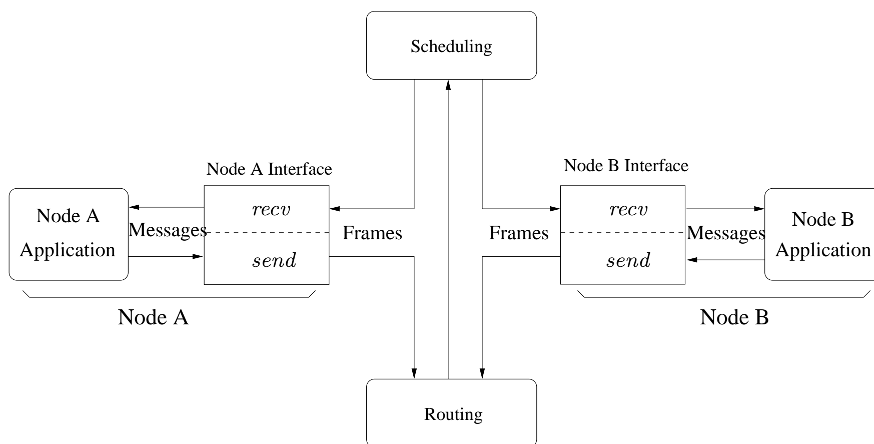

Figure 2: *GeNoC*: A generic network on chip model

generated from parameters *pms* defined on an arbitrary domain *GenParams* and function *NodeSetGen*. Valid parameters are recognized by predicate *ValidParamsp* and constitute the generating base for *NodeSet*. The functionality of *NodeSetGen* is as follows:

$$NodeSetGen : GenParams \rightarrow \mathcal{P}(GenNodeSet) \qquad (2)$$

These functions are valid if, for all parameters recognized by predicate *ValidParamsp*, every element produced by function *NodeSetGen* belongs to domain *GenNodeSet* (*i.e.* satisfies predicate *NodeSetp*):

PROOF OBLIGATION 1. **Definition of *NodeSet*.**

```
(defthm nodeset-generates-valid-nodes
  (implies (ValidParamsp pms)
           (NodeSetp (NodesetGenerator pms))))
```

Finally, we need to prove that, for each particular instance of predicate *NodeSetp*, any sublist of a valid list of nodes is also a valid list of nodes

PROOF OBLIGATION 2. **Sublists of Valid Node Lists.**

```
(defthm subsets-are-valid
  (implies (and (NodeSetp x) (subsetp y x))
           (NodeSetp y)))
```

Functions `NodesetGenerator`, `ValidParamsp` and `NodeSetp` are constrained to satisfy the given properties.

2.3.2 Route Validity

A route r is correct according to some missive m if (1) the first element of r equals the origin of m; (2) the last element of r equals the destination of m; (3) each node of r is a member of the set *NodeSet* of the existing nodes. The length of any route must be greater than 2. Among these properties, one only depends on *NodeSet*. To avoid free variables, we state it in a separate predicate. The other properties are defined as follows:

```
(defun ValidRoutep (r m)
  (and (equal (car r) (OrgM m))
       (equal (car (last r)) (DestM m))
       (<= 2 (len r))))
```

Function `CheckRoutes` takes a list of routes, a missive and the set `NodeSet`. It checks that any route of the list of routes satisfies `ValidRoutep` and is a subset of `NodeSet`.

```
(defun CheckRoutes (routes m NodeSet)
  (if (endp routes)
      t
    (let ((r (car routes)))
      (and (ValidRoutep r m)
           (subsetp r NodeSet)
           (CheckRoutes (cdr routes) m NodeSet)))))
```

Predicate `CorrectRoutesp` checks travels correctness according to missives, *i.e.* routes associated with some travel v satisfies predicate `CheckRoutes` for some missive m such that v and m have the same identifier and the same frame. We also check that the list of travels and the list of missives have the same length.

```
(defun CorrectRoutesp (V M NodeSet)
  (if (endp V)
      (if (endp M)
          t ;; len(M) = len(V)
        nil)
    (let* ((tr (car V))
           (msv (car M))
           (routes (RoutesV tr)))
      (and (CheckRoutes routes msv NodeSet)
           (equal (IdV tr) (IdM msv))
           (equal (FrmV tr) (FrmM msv))
           (CorrectRoutesp (cdr V)
                           (cdr M) NodeSet)))))
```

2.3.3 Generic Routing Function

The generic routing function takes two arguments: a missive list and the existing nodes. It returns a travel list. Its signature is the following:

```
(((Routing * *) => *))
```

The local witness of the encapsulate simply corresponds to routing in a bus. There is only one route made of the origin and the destination. In the following definition, functions IdM, FrmM, OrgM, DestM are the accessors of the various components of a missive: identifier, frame, origin, destination.

```
;; local witness
(local (defun route (M)
         (if (endp M)
             nil
           (let* ((msv (car M))
                  (Id (IdM msv))
                  (frm (FrmM msv))
                  (org (OrgM msv))
                  (dest (DestM msv)))
             (cons (list Id frm
                         (list (list org dest)))
                   (route (cdr M)))))))
(local (defun routing (M NodeSet)
         (declare (ignore NodeSet))
         (route M)))
```

The main constraint on function Routing states that it must satisfy predicate CorrectRoutesp.

Proof Obligation 3. **Routing Correctness**

```
(defthm Routing-CorrectRoutesp
  (let ((NodeSet (NodeSetGenerator pms)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp pms))
             (CorrectRoutesp (Routing M NodeSet)
                             M NodeSet))))
```

Another constraint checks that this function outputs a valid travel list. Predicate Vlstp checks that each travel is a tuple of the form *(id frm Routes)*, where *id* is a natural, *frm* is not equal to nil, and each route in list *Routes* contains at least two nodes.

Proof Obligation 4. **Type of function** *Routing*

```
(defthm Vlstp-routing
  (let ((NodeSet (NodeSetGenerator pms)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp pms))
             (Vlstp (routing M NodeSet)))))
```

We have shown the main constraints on function *Routing*. Some locals lemmas on the witness are necessary. There are two additional constraints. One that checks that function Routing outputs a true list. Another one checks that function Routing returns nil if the initial missive list is empty.

3. DETERMINISTIC ROUTING

In this section, we instantiate the previous generic routing module with a deterministic routing algorithm in a 2D mesh. This application has been treated in more detail [12]. We recall necessary elements to make the paper self-contained.

3.1 Mesh Node Definition

In a 2D mesh, a node is represented by a pair of coordinates on the X and Y axes. A pair of coordinates is recognized by predicate Coordinatep. A list of coordinates is recognized by predicate mesh-nodesetp.

Mesh parameters are the number of nodes in each dimension; they are recognized by predicate ValidParamsp2D. Let N_X and N_Y denote the number of nodes in the first and the second dimension. The node set, *i.e.* the set of coordinates from $(0, 0)$ to $((N_X - 1), (N_Y - 1))$, is generated by function mesh-nsgen. It is defined as follows.

Function $XGen(N_X, y)$ takes as arguments the number N_X of nodes in the first dimension and a constant y in the second dimension. It generates all admissible pairs for that particular y. Function mesh-nsgen computes the coordinates by applying function *XGen* to all values of y ranging from zero to $N_Y - 1$. To prove the main constraint on the node definition, we first prove that the generation on the X axis is valid, and use this fact prove that nodes generated on the Y axis are valid.

Theorem 1. **Mesh Nodes Validation.**

```
(defthm 2d-mesh-nodesetgenerator
  (implies (ValidParamsp2D pms)
           (mesh-nodesetp (mesh-nsgen pms))))
```

Once this theorem is proven, we check that the coordinates are a valid instance of the generic node definition by proving the following theorem:

```
(defthm check-2D-mesh-nodeset
  t
  :rule-classes nil
  :hints (("GOAL"
           :use
           (:functional-instance
             nodeset-generates-valid-nodes
             (NodeSetp 2D-mesh-NodeSetp)
             (NodeSetGenerator mesh-nodeset-gen)
             (ValidParamsp mesh-hyps))
           :in-theory
           (disable nodeset-generates-valid-nodes))))
```

This forces ACL2 to automatically generate and check all the constraints associated with the encapsulate event. This technique is systematically used to check compliance of modules. More details can be found in [12].

3.2 Dimension Order Routing

Dimension-order routing [4] is a deterministic routing scheme well-suited for uniform traffic distribution. The dimensions of the network are arranged in predetermined monotonic order and packets traverse dimensions in sequence. First, packets traverse the network in the lowest or highest dimension until no further move is needed in this dimension. Then, they go along the next dimension and so forth until they reach their destination. These algorithms are minimal.

Dimension-order routing in two-dimensional meshes is called XY routing (Fig. 3). The dimensions of the mesh are named X and Y. The chosen order is "X is less than Y". Packets travel along the X dimension completely and then along the Y dimension.

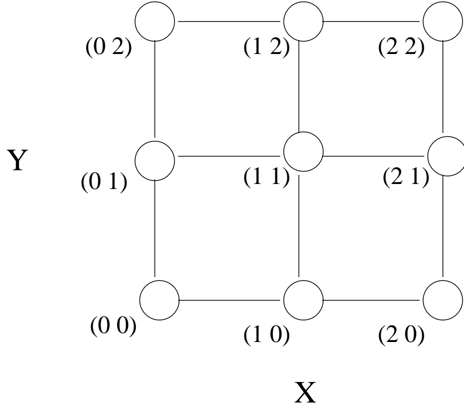

Figure 3: XY Routing

Let $s = (s_x, s_y)$ be a node containing a packet addressed to node $d = (d_x, d_y)$. In the XY algorithm, the X direction has higher priority. If the X of destination d is greater (resp. less) than the X of origin s, the next node is the node $(s_x + 1, s_y)$ (resp. $(s_x - 1, s_y)$) on the X-axis. Otherwise, the X's are equal and we compare the Y's: the next node is either $(s_x, s_y + 1)$ or $(s_x, s_y - 1)$ on the Y-axis. This algorithm is applied recursively to compute the route from a source to a destination. The measure is simply the distance between two nodes.

Definition 1. **XY Routing Algorithm**

```
(defun xy-routing (from to)
 (declare (xargs :measure (dist from to)))
 ;; from = (x_o y_o) dest = (x_d y_d)
 (if (or (not (coordinatep from))
         (not (coordinatep to)))
     nil
   (let ((x_d (car to))
         (y_d (cadr to))
         (x_o (car from))
         (y_o (cadr from)))
     (if (and (equal x_d x_o)  ;; x_d = x_o
              (equal y_d y_o)) ;; y_d = y_o
         ;; if the destination is equal to
         ;; the current node, we stop
         (cons from nil)
       (if (not (equal x_d x_o))  ;; x_d /= x_o
           (if (< x_d x_o) ;; decreasing x
              (cons from
                    (xy-routing (list (- x_o 1) y_o) to))
             ;; x_d > x_o
             (cons from
                   (xy-routing (list (+ x_o 1) y_o) to)))
         ;; otherwise we test the y-direction
         ;; y_d /= y and x_d = x_o
         (if (< y_d y_o)
             (cons
              from
              (xy-routing (list x_o (- y_o 1)) to))
           ;; y_d > y_o
           (cons
            from
            (xy-routing (list x_o (+ y_o 1)) to))))))))
```

This algorithm has been proven to conform with *GeNoC* [12]. This algorithm gives the priority to moves along the X-axis. We define function `yx-routing` to give the priority to the Y-axis. We do not detail the definition, which is mainly obtained by switching the order of the tests to first consider the Y-axis. This function is proven to conform with *GeNoC* by a simple *cut and paste* from the proof of function `xy-routing`.

4. ADAPTIVE ROUTING

4.1 Principles

Among minimal and adaptive algorithms, a well-known technique is to decompose a network into subnetworks. This is an abstract view of the network and subnetworks do not really exist. Depending on its destination, a packet uses one particular subnetwork. In each subnetwork, several minimal paths are available.

A typical example is the *double Y-channel routing algorithm.* This algorithm applies to a two-dimensional mesh where nodes are connected by one bidirectional channel in the X dimension and by two bidirectional channels in the Y dimension (Fig. 4). The network is cut into two subnetworks. Each subnetwork has one channel in the Y dimension. Each subnetwork uses the X channel in only one direction. The X^+ subnetwork (in dotted lines on Figure 4) uses this channel in the ascending X dimension. The X^- subnetwork (in plain lines on Figure 4) uses this channel in the decreasing X dimension. Let s and d be the source and destination nodes of a packet. Their coordinates along the X axis are noted s_x and d_x. If at some node the destination of a packet is on the right (*i.e.* if $d_x > s_x$), the packet uses the X^+ subnetwork; if the destination is on the left (*i.e.* if $d_x < s_x$), the packet used the X^- subnetwork. An arbitrary choice is made if $d_x = s_x$. In each subnetwork, several minimal paths are possible. A packet traverses the network through a single subnetwork.

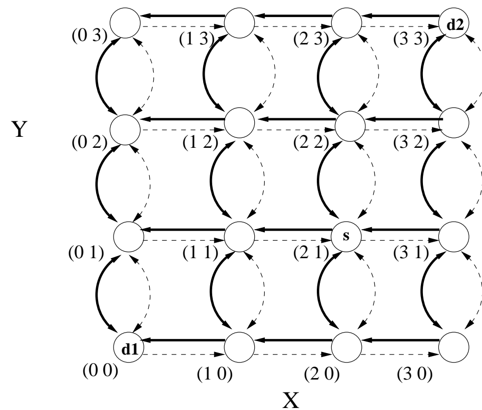

Figure 4: Double Y routing algorithm

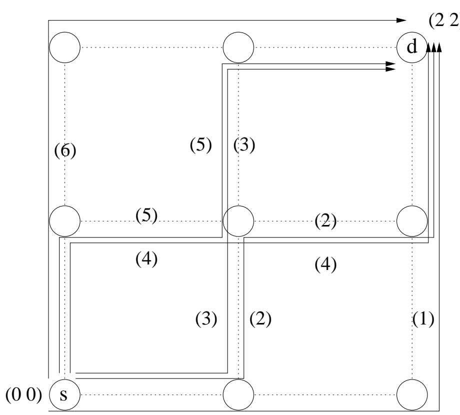

Figure 5: Minimal Possible Paths in a 3×3 mesh

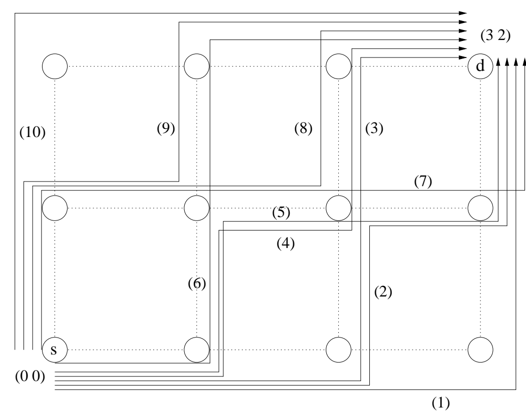

Figure 6: Minimal Possible Paths in a 4×3 mesh

EXAMPLE 1. *Let us consider Figure 4 and a packet,* p_1 *emitted at node* $s = (2\ 1)$ *and destined to node* $d_1 = (0\ 0)$*. As the destination of* p_1 *is on the left, it uses the* X^- *subnetwork. The possible paths are the following:*
(2 1) (1 1) (0 1) (0 0), (2 1)(1 1)(1 0)(0 0) or
(2 1) (2 0) (1 0) (0 0). Similarly, a packet, p_2*, emitted at node s and destined to node* $d_2 = (3\ 3)$ *uses one of the following routes of the* X^+ *subnetwork: (2 1)(3 1)(3 2)(3 3), (2 1)(2 2)(3 2)(3 3) or (2 1) (2 2) (2 3) (3 3).*

4.2 ACL2 Definition of Double Y

The generic routing function of *GeNoC* computes all possible routes between a source and a destination. The concrete function representing the double Y algorithm computes all possible minimal routes between two coordinates of a grid. In practice, only one route is taken. By proving all of them correct, *the* route taken by some message is correct.

For instance, between node *(0 0)* and node *(2 2)*, this algorithm proposes six possible routes (see Figure 5); between node *(0 0)* and node *(3 2)*, there are ten possible routes (see Figure 6).

We model the double Y algorithm as a function that applies alternatively the XY and the YX algorithms. First, note that for a node that has a common coordinate c with destination d, no choice is possible. In that case, the minimal path is to perform moves along the corresponding axis. Only nodes that have no common coordinate with the destination offer a choice between the two algorithms (XY or YX).

Our representation of the double Y algorithm is mainly performed by function `dy1`. We assume that `dy1` is called in a context where the origin and the destination have no common coordinate. This function takes the following input arguments:

- A list of *sources*. A *source* is simply a node at which a choice is possible. Initially, this list contains the initial (source) node.
- The final destination.
- A *flag* telling which algorithm should be applied at the current step. The flag is a Boolean. If it is true, one should apply the XY algorithm. Otherwise, the YX algorithm is applied.
- A list of *prefixes*. A *prefix* is a route where nodes have no common coordinate with the final destination. Prefixes are used to save intermediate route computations.

This function relies on three principal subfunctions:

- Function `extract-prefixes` extracts prefixes from a route and a destination
- Function `GetSources` extracts sources from a route
- To prove the termination of function `dy1`, we need a specific property on list `sources`. This property is defined by predicate `CloserListp`.

Function `extract-prefixes`. We first need a subfunction that extracts prefixes from a route r and a given destination d. This function is named `extract-prefixes` and takes as arguments a route r and a destination d. It is defined as follows[1]:

```
(defun extract-prefixes (r d)
  ;; compute prefixes suggested by route r
  ;; for destination d
  (if (or (not (coordinatep d))
          (not (2D-mesh-nodesetp r)))
      nil
    (if (endp r)
        nil
      (let* ((n1 (car r)) ;; n1 = first node of r
             (n1_x (car n1))
             (n1_y (cadr n1))
             (d_x (car d))
             (d_y (cadr d)))
        (if (or (equal n1_x d_x) (equal n1_y d_y))
            ;; nodes with common coordinate with d
            ;; do not allow any choice.
```

[1]Function `append-e-all` appends its first argument to each element of its second argument.

```
          nil
        (cons (list n1) ;; n1 is a prefix
              (append-e-all
               n1
               ;; all other prefixes start with n1
               ;; and recursive call
               (extract-prefixes (cdr r) d))))))))
```

Example 2. *From the route produced by the XY routing algorithm from node (0 0) to (3 2), this function extracts three prefixes:*
`(((0 0)) ((0 0) (1 0)) ((0 0) (1 0) (2 0)))`.

Function `GetSources`. We define function `GetSources` to extract a list of sources from a route r and a given destination d. This function is first defined by subfunction `GetSources1`:

```
(defun GetSources1 (r d)
  (if (or (endp r) (not (coordinatep d))
          (not (2d-mesh-nodesetp r)))
      nil
    (let* ((n1 (car r))
           (n1_x (car n1))
           (n1_y (cadr n1))
           (d_x (car d))
           (d_y (cadr d)))
      (if (or (equal n1_x d_x) (equal n1_y d_y))
          nil ;; n1 shares a coordinate with d
        (cons n1 (GetSources1 (cdr r) d))))))
```

The last nodes of prefixes are always sources and we append prefixes with routes computed from sources. Thus, a source appears both in a prefix and in a new route. To avoid the introduction of duplicates, we remove the first node of a route before extracting sources from it.

```
(defun GetSources (r d)
  ;; first node removed because already computed.
  ;; (it is already in the prefix)
  (GetSources1 (cdr r) d))
```

Example 3. *From the route produced by the XY routing algorithm from node (0 0) to (3 2), function* `GetSources` *produces two sources: nodes (1 0) and (2 0). Node (0 0) is already part of prefixes (see Example 2). It is not duplicated as a source.*

Predicate `CloserListp`. Finally, to prove the termination of our main function, we need that list `sources` be made of successive nodes, *i.e.* the distance to the destination from an element of sources and its successor decreases by 1. This is expressed by predicate `CloserListp` defined as follows:

```
(defun CloserListp (r d)
  ;; recognizes r as a list such that any higher
  ;; position gives a nodes that is closer to d
  ;; More exactly, the distance decreases by 1
  ;; if the position increases by 1,
  (if (or (endp r) (endp (cdr r)))
      t
    (and (equal (dist (cadr r) d)
                (1- (dist (car r) d)))
         (closerlistp (cdr r) d))))
```

Function `dy1`. The definition of `dy1` is given below. The measure is the distance between nodes. To prove the termination some hints are required. They are mostly used to disable some rules of the arithmetic library to avoid loops in the rewriter. We also need 7 additional lemmas. They show that the distance between nodes decrease by one at each application of the XY or the YX routing algorithm. The first branch of the conditional statement (lines 04 to 08) simply gets rid of bad inputs and ensures that list `sources` gives nodes that get closer to the destination. The first branch of the `cond` statement (lines 16 to 19) stops the algorithm because we have reached a node that shares a coordinate with the destination. The `let` statement at line 21 computes a new route by applying either the XY routing algorithm (lines 24 to 35) or the YX algorithm (lines 36 to 44). These two computations are similar, we only detail the XY case. We compute (line 26) a route from the first element of `sources` (line 10) and the destination. We append (line 30) this route with the first prefix (line 09) of list `prefixes`. This new route is `cons`'ed to a recursive call of `dy1`. The arguments of this call are the sources extracted from the new route (line 31), the destination and the negation of the flag (line 32), a new list of prefixes. The latter is obtained by adding the prefix used in the computation of the new route to every prefix suggested by this new route. Finally (lines 45 to 47), the routes computed in the `let` statement are added to the application of `dy1` to the remaining sources.

```
01 (defun dy1 (sources d flg prefixes)
02   (declare (xargs :measure
03                   (dist (car sources) d)))
04   (if (or (endp sources)
05           (not (CloserListp sources d))
06           (not (2d-mesh-nodesetp sources))
07           (not (coordinatep d)))
08       nil
09     (let* ((prefix (car prefixes))
10            (s (car sources))
11            (s_x (car s))
12            (s_y (cadr s))
13            (d_x (car d))
14            (d_y (cadr d)))
15       (cond
16        ((or (equal s_x d_x) (equal s_y d_y))
17        ;; if one coordinate has been reached,
18        ;; we stop
19         nil)
20        (t
21         (let
22          ((routes
23            (cond
24             (flg
25             ;; last was yx, next is xy
26             (let ((suffix (xy-routing s d)))
27             ;; the new route is made of the prefix
28             ;; and the new suffix of the route.
29               (cons
```

```
30               (append prefix suffix)
31               (dy1 (getSources suffix d)
32                    d nil ;; next is YX
33                    (append-l-all²
34                    prefix
35                    (extract-prefixes suffix d))))))
36             (t ;; last was xy-routing, next is yx
37              (let ((suffix (yx-routing s d)))
38               (cons (append prefix suffix)
39                     (dy1 (getSources suffix d)
40                      d t ;;next is XY
41                      (append-l-all
42                      prefix
43                      (extract-prefixes suffix d))))
44        )))))
45        (append routes
46                (dy1 (cdr sources)
47                     d flg prefixes)))))))))
```

We add the cases where no choice is possible. This defines function `dy` below:

```
(defun dy (s d)
  (if (and (coordinatep s) (coordinatep d)
           (or (equal (car s) (car d))
               (equal (cadr s) (cadr d))))
           (list (xy-routing s d))
      (append ;; flag is true,we start with xy
              (dy1 (list s) d t nil)
              ;; flag is false, we start with yx
              (dy1 (list s) d nil nil))))
```

EXAMPLE 4. *Let us consider the computation of all routes between nodes (0 0) and (3 2) in a 4 × 3 mesh. Let us consider that the flag is true. List* **sources** *contains source node s. List* **prefixes** *is initially empty. The first iteration produces route $r_{xy,0}$ by the application of the XY algorithm (function xy-routing) to node s (see Figure 7). This route suggests two new sources: nodes s_1 and s_2. The new prefix contains a route made of a single node (s) and two partial routes: one from s to s_1, and one from s to s_2.*

The algorithm is recursively applied to s_1, and then to s_2. The application to s_1 produces three routes (see Figure 8). The first route is obtained by the addition of s to the result of yx-routing(s_1, d), the second route is obtained by the addition of s and s_1 to the result of xy-routing(s_3, d), and the third route is obtained by the addition of s, s_1 and s_3 to the result of yx-routing(s_4, d). The application to s_2 produces the last two routes (see Figure 9). The first route is obtained by the addition of s and s_1 to the result of yx-routing(s_2, d), the second route is obtained by the addition of s, s_1 and s_2 to the result of xy-routing(s_4, d).

In the case of the flag is initially set to false, this algorithm produces four routes. We obtain all routes between s and d.

We complete the definition to match the signature of the generic routing function.

[2] Function `append-l-all` appends its first argument (which is a list) to every element of its second argument.

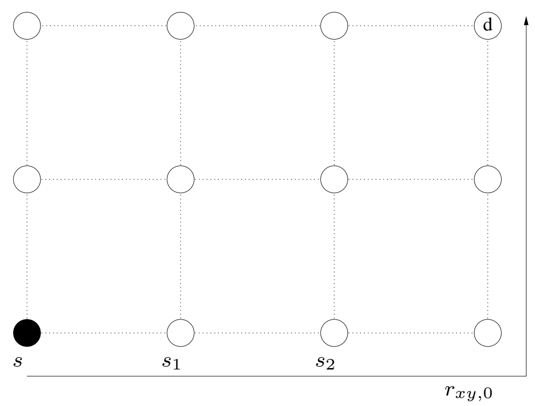

Figure 7: First iteration of the algorithm: application of *xy*-routing(s, d)

```
(defun DoubleYRouting1 (Missives)
  (if (endp Missives)
      nil
    (let* ((miss (car Missives))
           (org (OrgM miss))
           (dest (DestM miss))
           (Id (IdM miss))
           (frm (FrmM miss)))
      (cons (list id frm (dy org dest))
            (DoubleYRouting1 (cdr Missives))))))

(defun DoubleYRouting (Missives NodeSet)
  (declare (ignore NodeSet))
  (DoubleYRouting1 Missives))
```

4.3 Verification

The main property to verify is that predicate `CorrectRoutesp` holds for our double Y algorithm. We can decompose the proof in two subproperties: all routes produced by `dy(s,d)` (1) start with s and end with d; (2) are subsets of the set of existing nodes. The proof of the latter is rather easy. We focus on the proof of the former.

Function `GetSources`. This function must return a list of nodes that do not share a coordinate with the destination. We formalize this property by predicate `no-common-coordinate` defined as follows:

```
(defun no-common-coordinate (lst d)
  ;; nodes in lst do not share a coordinate with d
  (if (endp lst)
      t
    (let ((n (car lst))) ;; pick a node
      (and (not (equal (car n) (car d))) ;; x-coord.
           (not (equal (cadr n) (cadr d)));; y-coord.
           (no-common-coordinate (cdr lst) d)))))
```

We prove that function `getSources` satisfies it:

```
(defthm no-common-coordinate-getSources1
  (no-common-coordinate (getSources1 lst d) d))
```

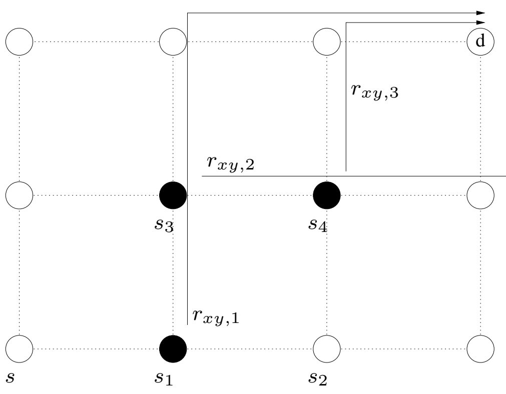

Figure 8: Iterations 2,3 and 4 : application of yx-routing(s_1, d), xy-routing(s_3, d) and yx-routing(s_4, d)

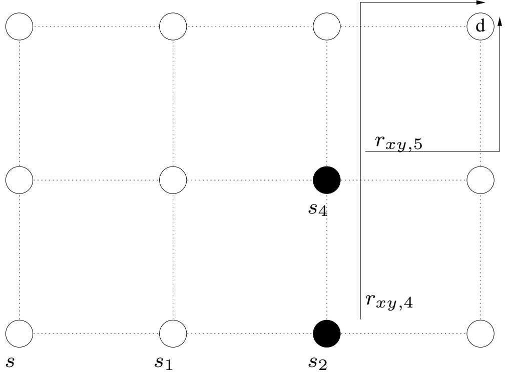

Figure 9: Iterations 5 and 6 : application of yx-routing(s_2, d) and xy-routing(s_4, d)

This proof is fully automatic in ACL2 and do not require any additional lemmas.

Prefixes and Sources. To prove that function `dy` produces routes that starts with the right origin, we need to prove that functions `xy-routing` and `yx-routing` produce correct routes. We also need a similar property for `prefixes` and `sources`. Functions `xy-routing` and `yx-routing` have already been proven to conform with *GeNoC*. Therefore, we already know that they produce correct routes. The first node of any prefixes must start with the origin. This is expressed by the following function:

```
(defun inv-prefixes (prefixes s)
  (if (endp prefixes)
      t
    (and (equal (caar prefixes) s)
         (inv-prefixes (cdr prefixes) s))))
```

We have a similar property for the list of sources:

```
(defun inv-sources (sources s)
  (if (endp sources)
      t
    (and (equal (car sources) s)
         (inv-sources (cdr sources) s))))
```

The relation between these two invariants is that the second one holds only if the list of prefixes is empty. We group them in the following definition:

```
(defun inv (prefixes sources s)
  (if (endp prefixes)
      (inv-sources sources s)
    (inv-prefixes prefixes s)))
```

This invariant is used in the next inductive proof. This invariant is trivially satisfied when list `prefixes` and `sources` are both empty (base case).

Main Lemma. The main lemma proves that function `dy1` satisfies predicate `all-validroutep`:

```
(defthm validroutep-apply-DoubleY-all-nodeset
  (implies (and (all-2d-mesh-nodesetp prefixes)
                (coordinatep d) (coordinatep s)
                (not (equal s d))
                (no-common-coordinate sources d)
                (consp sources)
                (2d-mesh-nodesetp sources)
                (inv prefixes sources s))
           (all-validroutep
            (dy1 sources d flg prefixes) s d)))
```

This lemma is proven using the induction scheme suggested by function `dy1`. To complete the proof we need to show that invariant `inv` is preserved in the induction step. This amounts to the proof of properties expressing the link between this invariant and functions `append-l-all`, `getSources`, and `extract-prefixes`. In the case of the XY routing algorithm, this is expressed as follows:

```
(defthm inv-append-l-all-xy-routing
  (implies (and (inv prefixes sources s)
                (no-common-coordinate sources d)
                (coordinatep s) (coordinatep d)
                (consp sources)
                (all-2d-mesh-nodesetp prefixes)
                (2d-mesh-nodesetp sources))
           (inv (append-l-all
                  (car prefixes)
                  (extract-prefixes
                      (xy-routing (car sources) d)
                      d))
                (getSources1
                  (cdr (xy-routing (car sources) d))
                  d)
                s)))
```

We prove a similar lemma for function `yx-routing`. The two proofs rely on the correctness of the XY and YX algorithms, as well as additional lemmas expressing the relations between more basic functions (like `apppend-e-all`) and the invariant.

Table 1 summarizes data about the ACL2 proof. The size is measured by the number of lines of code. Results regarding execution time are given in seconds. These results were obtain on an Intel Dual Core 2400 machine with 2GB of memory. The row regarding function `dy` includes also the definition and the validation of function `yx-routing`. The latter reuses part of the book developed for function `xy-routing`. The proof time for function `dy` is mainly due to arithmetic reasoning in the termination proof.

	`defun`	`defthm`	size	time
Mesh NodeSet	8	6	120	0.55
`xy-routing`	7	44	520	3.8
`dy`	21	84	1200	67.6

Table 1: Data for the Double Y Algorithm

5. CONCLUSION AND FUTURE WORK

We have presented the specification and the validation of an adaptive routing algorithm for a 2D mesh. The adaptive algorithm makes use of definitions and theorems developed for the deterministic case. Following "The Method" ([7]) the definition and the validation of the double Y algorithm could be developed in about two weeks by an ACL2 expert familiar with routing algorithms in two-dimensional meshes and the *GeNoC* framework.

Our current treatment of adaptive routing algorithms considers the computation of *all* possible routes. If this works for minimal algorithms, it might not be feasible for non-minimal algorithms. We plan to extend *GeNoC* with a global notion of the network state and to study the application to non-minimal algorithms. Finally, the current *GeNoC* definition is very abstract and very simplified. Successive, proven correct refined models are needed before reaching the level of details of an implementation specification. We plan to extend our work in this direction.

As noticed by two anonymous reviewers, our use of constrained functions and functional instantiation raises an important issue. The issue is that ACL2 has no event whose success means "this concrete system is a valid instance of this generic system". Currently, a functional-instantiation is successful if ACL2 can prove (1) that the instantiated lemma implies the original goal, and (2) that the concrete functions satisfy the given constraints. One could imagine the following "improvement" of ACL2. If ACL2 cannot prove that the constraints are satisfied, it will try to solve the original goal. In our case, this would mean proving 't' and would obviously succeed. Our trick of proving "t" would have nothing to do with checking the valid instantiation of a generic system. In any case, we would call for the development of a new event that will only check that concrete functions satisfy the constraints [3].

Acknowledgements

The author would like to thank the anonymous referees for their constructive comments. We are especially grateful to the reviewer who provided two pages of apposite comments.

[3]In release 3.2.1 of ACL2, experimental macro definitions (see file defspec.lisp in book directory make-event) are being developed by S. Ray and M. Kaufmann to solve this issue.

6. REFERENCES

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

[2] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A Generic Model for Formally Verifying NoC Communication Architectures: A Case Study. In *Proc. of First International Symposium on Networks-on-Chip (NOCS'07)*, pages 127–136, Princeton, NJ, USA, 7-9 May 2007. IEEE.

[3] 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.

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

[5] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Rădulescu. Deadlock Prevention in the Æthereal protocol. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 345–348, 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] M. Kaufmann, P. Manolios, and J. S. Moore. *ACL2 Computer Aided Reasoning: An Approach.* Klulwer Academic Press, 2000.

[8] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Eleventh International Conference on Automated Deduction (CADE'92)*, volume 607 of *LNAI*, pages 748–752, Saragota, NY, June 1992. Springer-Verlag.

[9] J. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. In *34th Design Automation Conference (DAC'96)*, pages 178–183, 1997.

[10] J. Schmaltz. *Une formalisation fonctionnelle des communications sur la puce.* PhD thesis, Joseph Fourier University, Grenoble, France, January 2006. In French. A partial translation is available upon request to the first author.

[11] J. Schmaltz and D. Borrione. A Generic Network on Chip Model. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 310–325, Oxford, UK, August 2005. Springer-Verlag.

[12] J. Schmaltz and D. Borrione. Towards a Formal Theory of On Chip Communications in the ACL2 Logic. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications, part of FloC'06*, Seattle, Washington, USA, August 14-15 2006. ACM.

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