ABSTRACT
Complex systems-on-chips (SoCs) are built as the assembly of pre-designed parameterized components. The specification and validation of the communication infrastructure becomes a crucial step in the early phase of any SoC design. The Generic Network-on-Chip model (GeNoC) has been recently proposed as a generic specification environment, restricted to safety properties. We report on an initial extension of the GeNoC model with a generic termination condition and a generic property showing the prevention of livelock and deadlock. The latter shows that all messages injected in the network eventually reach their destination for all possible values of network parameters like topology, size of the network, message length or injection time. We illustrate our initial results with the validation of a circuit switching technique.

Categories and Subject Descriptors
B.7.2 [Integrated Circuits]: Design Aids—verification

General Terms
algorithms, verification

Keywords
liveness, networks-on-chips, formal methods

1. INTRODUCTION
Integration capabilities of chip technologies enable the production of Multi-Processors Systems-on-Chips (MPSoCs) composed of several processing and memory cores, as well as peripherals and I/O devices. The design of such complex systems follows a platform-based approach where a new MPSoC is built as the assembly of pre-designed and parametric components – called Intellectual Properties (IPs) – according to a generic architecture [18]. Communications become a bottleneck. To meet system requirements networks-on-chips (NoCs) emerge as an adequate communication infrastructure [1]. To handle the complexity of modern MPSoCs initial design phases must begin at higher levels of abstraction, while keeping a link with the final Register Transfer Level (RTL) implementation [15].

As communications are becoming dominant to the overall correctness and performance of an MPSoC, their formal validation at their initial design phase will soon become mandatory. The Generic Network-on-Chip (GeNoC) model [14, 2] offers a general environment to reason about high-level and parametric descriptions of NoCs. It has been implemented in the ACL2 logic [13] and applied to several case-studies. GeNoC is a function formalizing the interactions between three essential constituents: interfaces, routing algorithms, and scheduling policies. Each one of them is generic in the sense that it is not given a particular definition but characterized by a set of proof obligations or constraints. GeNoC is proven to satisfy a global correctness theorem, the proof of which depends on the proof obligations only. It can therefore be instantiated for any definition of the constituents which satisfy the proof obligations. Verifying a particular NoC reduces to discharging these instantiated constraints for the NoC constituent. The verification methodology proceeds by (1) giving a concrete definition to each one of the constituents; (2) the corresponding constraints are automatically generated; (3) proving that each concrete definition satisfies the corresponding constraints; (4) it automatically follows that the concrete network satisfies the instantiated global theorem.

In its current version, this global theorem states that every message received at some destination node was actually issued at a valid source node, and followed a valid path to reach its expected destination. We report on work in progress towards an extension of this theorem that would guarantee that eventually all messages reach their destination. The theorem would include an upper bound on the time needed to “evacuate” all these messages and would prevent the network from any deadlock state.

The contributions of this paper are (1) an extension of the definition of GeNoC with a generic termination condition, (2) a generic property showing that all messages injected in the network reach their destination, and (3) the application of this new model and property to prove deadlock prevention of a circuit switching technique.

In the next section, we briefly present the necessary knowledge about the GeNoC model. Section 3 presents our ex-
tended definition, the termination condition, and a generic deadlock prevention theorem. Section 4 instantiates this new generic definition with a circuit switched network. We prove deadlock prevention in Section 4.3. We discuss related work in Section 5 before concluding the paper in Section 6.

2. THE GENERIC NOC MODEL
The Generic NoC (GeNoC [14, 2]) model represents the transmission of messages from their source to their destination on a generic communication architecture with an arbitrary network characterization (topology and node interfaces), routing algorithm, and switching technique. The model is composed of a collection of functions together with their characteristic constraints. The main function is recursive and each recursive call represents one step of simulation. Such a step defines our time unit.

2.1 The Generic NoC Model
The model considers a set of addresses that can emit or receive messages. A message $m$ is uniquely identified by a natural number $m.id^1$. To analyze a message, we associate it with its origin $m.org$, its current address $m.curr$, its destination $m.dest$, its content $m.msg$, and the execution step $m.time$ at which it is emitted. An address together with its content constitutes a state element of the global network state.

Function GeNoC (Fig. 1) takes a list of messages to be sent at different execution steps and produces the list of messages that have reached their destination and a list containing those that are still "en route" or never left their source.

- Network access control: Function $r4d$ produces a list of traveling messages which are injected in the network and a list of delayed messages.

- Routing: The traveling messages are given to function Routing, which computes routes from the current to the destination address for each message.

- Scheduling: Function Scheduling represents the execution of one network simulation step. Using the routes produced by function Routing and considering the current global state, it moves – or not – a message and updates the global state accordingly. Messages that have reached their destination constitute the list Arrived, and the rest constitutes the list EnRoute.

- Recursion: Functions $r4d$, Routing, and Scheduling are combined together. The lists enroute and delayed constitute the main argument of a recursive call to GeNoC. Arrived messages are accumulated after each recursive call. When function GeNoC terminates, the list Arrived contains all messages that have completed their path from their source to their destination; the list EnRoute contains all messages that have left their source but have not left the network; Delayed contains all messages that are still at their origin.

A pseudo-code for function GeNoC is given below. Function GeNoC takes as parameters the list of messages to be sent ($mlst$), the structure of the network, reduced to the set of its nodes ($NS$), a finite number of attempts ($att$). Function GeNoC also takes as input the list of arrived messages ($arr$, originally empty), the current state of the network ($ntkst$), and the current time ($time$). If no attempt is left, GeNoC stops and returns a pair composed of the arrived ($arr$), and the delayed ($mlst$) messages. Otherwise, every recursive call processes a list of messages, where some are waiting at their source, and some are traveling in the network. For each traveling message produced by functions $r4d$ and Routing, function Scheduling computes the list of the arrived messages ($arr'$), the list of messages that are still traveling in the network ($mlst'$), the remaining attempts ($att'$), and a new state ($ntkst'$). The recursive call processes the traveling messages together with the messages delayed by $r4d$ ($D$). Time is incremented by 1.

```
func GeNoC (mlst, NS, att, arr, ntkst, time) =
  if SumOfAttempts(att)=0
    then list(arr, mlst) ;; mlst = en route + delayed
  else
    let (TR D) = R4D(mlst,time) in
    let (mlst' arr' att' ntkst') =
      Scheduling(Routing(TR, NS), att, ntkst)
      in GeNoC (union(mlst', D), NS, att',
        union(arr, arr'), ntkst', time+1)
```

2.2 Functional correctness
The functional correctness of GeNoC is expressed as a theorem stating that all arrived messages can be matched to a unique message of the input list $mlst$. In other words, if a message has arrived at a node $d$ it was actually emitted at a valid source node $s$ with the same content and $d$ was the expected destination.

**Theorem 1.** $\forall r \in Arr, \exists ! m \in mlst : r.id = m.id \land r.msg = m.msg \land r.dest = m.dest$

The proof of this property only depends on the proof obligations associated to each function of GeNoC. We focus on functions Routing and Scheduling. The main proof obligations associated to these functions are the following:

- **Routing:** given a current address $c$ and a destination address $d$, function Routing computes a route such that its first element is $c$, the last element is $d$, and all elements belong to the set of valid addresses.
Scheduling:

1. lists EnRoute and Arrived are disjoint subsets of the routed messages given as input to Scheduling

2. function Scheduling consumes at least one attempt

The sum of attempts is a specific decreasing measure for GeNoC. When GeNoC terminates, it is possible that messages are still en route even though these messages are able to progress through the network. These en route messages may also be blocked and cannot make any further progress, i.e., the network is in a deadlock state.

In the next section, we detail a modified definition of GeNoC such that if GeNoC terminates and the list EnRoute is empty, there is no deadlock. In the remaining sections, we instantiate this new generic definition for a circuit switching technique.

3. EXTENDED MODEL

3.1 New definitions

Function GeNoC is divided in a top level function (named GeNoC) which formats inputs arguments for a core recursive function (named GeNoCt). We first describe function GeNoCt (see Fig. 2).

Function GeNoCt takes as arguments a list of messages, the set of nodes, the measure argument, an accumulator for arrived messages, the simulation step, and the network state. It returns a list of arrived messages and the list containing the en route and delayed messages. Predicates legal-measure and scheduling-assumptions (lines 11 and 14) are concerned with termination, which is explained in Section 3.2 below. Function rfd determines the set of messages that can be in the network at the current time (line 8). These messages can either be injected in the network or are already traveling in the network. Function Routing computes routes for these messages (line 10), and function Scheduling determines which of them can make a hop or have reached their destination (line 17).

Function GeNoC is given below. It takes as arguments a list of pending communications (mlst) and two parameters. The first one is used to generate the set of valid nodes (NS). The second one is used to generate the state of the network (ntkst). Function GeNoC returns two lists: a list of results (the arrived messages) and a list of en route messages.

Function GeNoC first check that parameters p1 and p2 are well-formed. If this is not the case, it returns two empty lists. Otherwise it calls function GeNoCt after building the initial values for its input arguments. Finally, function GeNoCt returns the lists of arrived and en route messages.

```
(defun GeNoC (mlst p1 p2)
  (if (ValidStateParamsp p1 p2)
      (let* ((NS (NodeSetGenerator p1))
             (st (StateGenerator p1 p2))
             (ntkst (GenInitState trs st))
             (v (routing mlst NS))
             (meas (InitMeas v NS ntkst)))
        (mv - let (arrived enroute)
                (GeNoCt mlst NS meas
                        nil nil '0 ntkst)
                (mv (computeresults arrived)
                     enroute)))
      (mv nil nil)))
```

This new definition of GeNoC is proven to satisfy the same
Figure 2: New definition of \textit{GeNoC}_t

generic correctness theorem than the previous one.

3.2 Termination
Function \textit{GeNoC}_t (see Fig. 2) terminates in three different ways: there is no message left to be processed (lines 2–4), the measure has reached an exit value (lines 11–13), or the network is in a deadlock state (lines 23–25). We consider the second case.

The decreasing measure declared for function \textit{GeNoC}_t is \((\text{acl2-count measure})\). Parameter \textit{measure} represents an upper bound to the \textit{evacuation time}, i.e., the number of steps that are necessary to inject and evacuate all messages in the input list \textit{mlst}. For example, if one instantiates function \textit{Scheduling} such that at each step at least one message arrives at its destination, then, \textit{measure} could be the number of messages on the network.

Predicate \textit{legal-measure} defines the halting condition for parameter \textit{measure}. Function \textit{Scheduling} is responsible for producing a measure that is decreasing as long as predicate \textit{legal-measure} holds. In general, this is not always possible. If no progress can be made (e.g., if all buffers of the network are full), then no correct representation of the evacuation time can be decreased. Predicate \textit{scheduling-assumptions} solves this issue. It must be instantiated in such a way that if it holds, function \textit{Scheduling} must be able to make progress and to provide a decreased measure. If this predicate does not hold, a deadlock state is reached. An example instantiation is given in section 4.1.

A new proof obligation is added. It states that if the scheduling assumptions hold and if the current measure is legal, the scheduler must be able to provide a new measure that is smaller than the current one.

\begin{verbatim}
(defthm measure-decreases
  (implies
    (and
      (legal-measure measure v NS ntkst)
      (scheduling-assumptions v NS ntkst))
    (0< (acl2-count
      (mv-nth 2 ;; get new measure
        (scheduling v NS ntkst))))))
\end{verbatim}

3.3 Deadlock prevention
The main proof obligation of \textit{Scheduling} states that the intersection of \textit{Arrived} and \textit{EnRoute} is empty. This implies that messages either never left their source, are en route in the network, or have reached their destination.

If \textit{EnRoute} is not empty, \textit{GeNoC}_t either terminated because the provided measure was not legal, or because the scheduling assumptions were not true. If we can prove that (1) the scheduling-assumptions are always true, and (2) the measure provided by function \textit{Scheduling} is always legal, then the only way for \textit{GeNoC}_t to terminate is when the input list is empty. If the input list is empty, \textit{EnRoute} is empty as well. Each injected message has reached its destination. No deadlock has occurred.
A typical form of a deadlock prevention theorem is:

\[
\text{defthm deadlock-free-genoc} \\
\text{(implies 'property on network state' } \\
\text{ (equal } \\
\text{ (mv-nth 1 (GeNoC messages p1 p2)) nil) \)}}
\]

This is a very general theorem. No assumption is made on the topology, and the size of the network, message length and injection time are left uninterpreted.

The theorem suggests the following proof methodology. Define a property \( p \) such that (1) \( p \) implies the scheduling-assumptions (line 14 in Fig. 2), and (2) \( p \) is inductive for GeNoC, i.e., if it holds initially, it holds after each recursion step. Such a property \( p \) together with a proof that the measure provided by Scheduling is always legal is sufficient to prove the deadlock prevention theorem.

### 4. CIRCUIT SWITCHING TECHNIQUE

Switching techniques determine how messages travel through a network. A switching technique is concerned with one or more types of resources, such as buffer space or channel bandwidth. Switching entails among others allocating resources to messages, determining where to send messages based on the sets of routes provided by the routing algorithm and resolving contention [3]. Contention occurs when two messages need the same resources. A message is blocked if it requires an unavailable resource. We consider switching techniques where no messages are dropped. If for some reason a message is blocked, it must be stored at its current position.

Circuit switching (CS) tries to establish a connection between the origin \( o \) and the destination \( d \) of a message before actually sending it. An established connection is called a circuit, which is also the type of resources CS deals with. A route is CS-possible if a circuit can be established for that route, i.e., if all nodes of the route have available space to store the message. Figure 3 gives an example of how two messages can traverse through a network with CS. Initially, both routes are CS-possible. However, only one circuit can be established at a time, since the routes intersect. Thus, one message is blocked until completion of the other.

A circuit is established by propagating a request from \( o \) to \( d \). Each node of the route receives a request to deny messages of any node other than \( o \). A node that acknowledges this request is called booked. A node will always acknowledge a request, unless it is booked or for some reason unavailable. This means that once the entire route is booked, a message can be sent with guaranteed throughput. After completion of all communications the source node sends a "torn-down" packet to release the circuit. A route is CS-possible if all its nodes are not booked and have an empty buffer.

We consider an abstract version of CS which computes a set of possible routes that do not intersect and schedules them all at once, i.e., in one scheduling step. If a message is scheduled, it is propagated through its circuit and arrives at its destination in the same step. This means that any scheduled message is removed from the network.

We consider a two-dimensional mesh, for which function Routing is instantiated with xy-routing. A message is routed along the X-axis before moving along the Y-axis. Our instance of function \( r4d \) injects all messages at the initial simulation step. All nodes have *num-of-buffers* buffers. Each buffer can store one packet of any size.

**Example 1.** Consider Fig. 4. Initially, the scheduler gets a message list with three messages \((1, 2, 3)\). The routes of messages 1 and 2 are both possible and do not intersect, therefore they are both scheduled. Message 3 cannot be scheduled since its route intersects with message 1. It is thus scheduled in the next scheduling step.

Our instance of GeNoC, for CS is given by function simple-genocy (Fig. 5). Theorems in the next sections are proved for this instantiated function. We made as few instantiation-specific assumptions as possible. All such assumptions are mentioned explicitly.

### 4.1 Instantiation of the generic measure

Argument "measure" is defined as a list where each element corresponds to the length of the route of a message. The decreasing measure declared for function \( ct\text{-scheduler} \) is the sum of the elements of this list. This sum is computed by function \( \text{sum-of-lst} \), which is not spelled out for brevity.

Function get-route returns the route computed by function routing. Function RouteLengths builds the measure argument as follows:

\[
\text{(defun RouteLengths } \text{(mlst) } \\
\text{(if } \text{(endp mlst) nil } \\
\text{ (cons (len (get-route (car mlst)))) (RouteLengths (cdr mlst)))))
\]

Function \( ct\text{-legal-measure} \) returns true only if the measure is the sum of the list of route lengths. Since it instantiates the generic function \( \text{legal-measure} \) (section 3.2), it takes the same parameters.

\[
\text{(defun ct\text{-legal-measure } \text{(meas mlst ntkst) } \\
\text{(equal meas } \\
\text{(sum-of-lst (RouteLengths mlst))}))}
\]

![Figure 3: The arrows show the routing of the messages in these nodes.](image)

![Figure 4: Example of CS.](image)
We define an ordering over such lists. Function `elts-<=` recognizes two lists \( x \) and \( y \) such that all elements of \( x \) are pairwise less or equal to the elements of \( y \).

\[
\text{(defun elts-<= (x y)}
\text{  (if (endp x)(endp y)}
\text{    (and (natp (car x)) (natp (car y))}
\text{      (<= (car x) (car y))}
\text{      (elts-<= (cdr x) (cdr y))))}
\]

We prove that if two lists \( x \) and \( y \) are `elts-<=` and the first element of \( x \) is strictly less than the first element of \( y \), the sum of the elements of \( x \) is strictly less than the sum of the elements of \( y \):

\[
\text{(defthm smaller-car-implies-smaller-sum}
\text{  (implies (and (< (car x) (car y))}
\text{    (elts-<= x y))}
\text{  (< (sum-of-lst x) (sum-of-lst y))))}
\]

We now prove that the new measure – i.e., the new list of route lengths – is never increased by function `Scheduling`:

\[
\text{(defthm scheduled-routes-<=-original}
\text{  (let ((new (mv-nth 2 (ct-scheduler mlst}
\text{    ER Arr meas prev ntkst))}
\text{    (old (RouteLengths mlst))))}
\text{  (elts-<= new old))))}
\]

The instance of `scheduling-assumptions` for CS checks that there must exist a CS-possible route in the current state. Predicate `no-good-routes` is defined, which returns `t` if and only if there is no CS-possible route in \( mlst \). This is done by checking that for all messages in \( mlst \) there exists a node in the route that is full, which implies no circuit can be established. Predicate `ct-scheduling-assumptions` is then defined as `not (no-good-routes mlst ntkst))`.

If a route is possible for the first message – i.e., if the scheduling assumptions are satisfied – function `Scheduling` is proven to reduce the length of the route of this message.

\[
\text{(defthm good-route-implies-smaller-routes}
\text{  (let ((new (mv-nth 2 (ct-scheduler mlst}
\text{    ER Arr meas prev ntkst))}
\text{    (old (RouteLengths mlst))))}
\text{  (implies ('(car mlst) has possible route'}
\text{    (< (car new) (car old))))})
\]

Finally, ACL2 can prove the proof obligation for termination automatically:

\[
\text{(defthm good-route-implies-smaller-measure}
\text{  (let ((new (mv-nth 2 (ct-scheduler mlst}
\text{    ER Arr meas prev ntkst))}
\text{    (implies (scheduling-assumptions}
\text{    mlst ns ntkst)
\text{    (< (sum-of-lst new) (sum-of-lst}
\text{    (RouteLengths mlst))))})}
\]

4.2 Instantiation of scheduler

In our model of circuit switching, a message can be scheduled if its route does not intersect with the routes of the currently scheduled messages and if it is CS-possible.

Function `test_prev_routes` takes as parameters a route \( r? \) and a set of routes \( prev \). It returns `t` if it does not intersect with any route in \( prev \). Otherwise it returns `nil`. 

Figure 5: Instantiation of GeNoC\(_t\) for CS
(defun test_prev_routes (r? prev)
  (if (endp prev) t
    (and (no_intersectp r? (car prev))
      (test_prev_routes r? (cdr prev)))))

Function ct-test_routes takes as parameters a message m and the network state ntkst. It returns the route r? of m if and only if r? is CS-possible, i.e., if all nodes of the route have an empty buffer. Otherwise it returns nil.

Function ct-scheduler combines these functions. It keeps track of all scheduled routes in prev. It first tries to find a CS-possible route (line 8). If it has found a possible route, it checks whether the route intersects with any route of prev (line 11). If the route does not intersect, the message is scheduled which in effect means that it is removed from the network (line 19) and added to list EnRoute (line 22). Otherwise, it is delayed and added to EnRoute (line 22).

(defun ct-scheduler (mlst EnRoute Arrived measure prev ntkst)
  (if (endp mlst)
    (mv (rev EnRoute) (rev Arrived)
      (rev measure) ntkst)
    (let ((m (car mlst)))
      (mv-let (newntkst r?)
        ;; access data link layer
        (ct-test_routes ntkst m)
        (if (and
            r?
            (test_prev_routes r? prev))
          ;; if there is a possible route, then remove v and add it to prev
          (ct-scheduler (cdr mlst) EnRoute
                        (cons m Arrived)
                        (cons 0 measure)
                        (cons r? prev)
                        (replace-in-node (Orgv m)
                                          (FrmV m) nil newntkst))
          ;; otherwise the transaction is delayed
          (ct-scheduler (cdr mlst) (cons m EnRoute) Arrived
                        (cons (len (get-route (car TrLst)) measure) prev
                              newntkst)))))))

Function ct-scheduling is then simply defined as follows:

(ct-scheduler mlst nil nil nil ntkst)

4.3 Deadlock prevention theorem

Function cs-deadlockfree below considers a list of messages (mlst) and an initial network state (ntkst). It returns t if and only if it is possible to process all messages of mlst, i.e., if no deadlock is possible. Parameter n represents the number of messages that have been analyzed. It initially equals 0. If it gets larger than the length of list mlst, the recursion stops. Otherwise, we check that the n’th message and the next one are cs-deadlock free.

(defun cs-deadlockfree (n mlst ntkst)
  (if (not (in-range n mlst)) t
    (and (deadlockfree_r
          (nth n mlst) nil mlst ntkst)
         (cs-deadlockfree
          (1+ n) mlst ntkst)))))

The mutually recursive functions deadlockfree_r and ∀-deadlockfree_r define the CS deadlock free condition. Function deadlockfree_r takes as arguments a message to be analyzed (m), an accumulator of messages that have already been analyzed (m-acc, initially empty), a list of messages mlst, and the network state (ntkst). If first gets the route r (function get-route) of the message (line 2). Then, it extracts from mlst the list of messages, the routes of which intersect with route r. If message m has already been analyzed, a cycle is detected and the network is not free from deadlock (line 6). If the buffers of route r have available space, then a circuit can be created for m. Hence, at least one message can make progress. There is no deadlock. Note that in order to check whether a route is possible we only need to check the cdr of the route (lines 4 and 7). Indeed, a route is possible, even if the current node is full. If the nodes of route r have no available space, function ∀-deadlockfree_r uses function deadlockfree_r to check that all messages that are blocking message m are free from deadlock. Message m is accumulated in m-acc.

(mutual-recursion
  (defun deadlockfree_r (m m-acc mlst ntkst)
    (let* ((r (get-route m))
           (mlst’ (get-mlst-route (cdr r) mlst)))
      (cond
        ((member-equal m m-acc) nil)
        ((has-empty-buffers (cdr r) ntkst) t)
        (t
          (∀-deadlockfree_r
           mlst’ (cons m m-acc) mlst ntkst)))))

(defun ∀-deadlockfree_r (mlst’ m-acc mlst ntkst)
  (if (endp mlst’) t
    (and (deadlockfree_r
          (car mlst’) m-acc mlst ntkst)
         (∀-deadlockfree_r
          (cdr mlst’) m-acc mlst ntkst))))

Example 2. Consider the examples in Fig. 6(a). When function deadlockfree_r is called for message 1, it first checks whether message 1 has a possible route. This is not the case, since its destination node is full. It therefore adds message 1 to m-acc and checks whether messages 2 is deadlockfree. Since message 2 has a possible route, it is deadlockfree and thus message 1 is deadlockfree as well.

In the situation depicted in Fig. 6(b), messages 1 and 2 are not deadlockfree. If deadlockfree_r is called for message 1 it
will accumulate message 1 in m-acc and call deadlockfreeₘ for message 2. To check deadlockfreedom of message 2 we must check deadlockfreedom of message 1. This message was accumulated in m-acc and therefore message 2 will be considered not deadlockfree. Thus message 1 is considered not deadlockfree as well.

Note that function deadlockfree checks for deadlockfreedom and not for a deadlockstate, which is a weaker property. The situation in Fig. 6(b) is not in a CS-deadlockstate, since message 3 can set up a circuit and arrive at its destination. Function deadlockfree still returns nil for this situation, because it is not deadlockfree.

Figure 6: Example of CS deadlockfreedom. Each node has one buffer.

The following lists are the measures for resp. cs-deadlockfreeₘ and ∨-cs-deadlockfreeₘ. Lexicographical ordering is used.

( list ( if ( member m mlst ) 0 1) (diff-size m-acc mlst 0 0) (list ( if ( subsetp mlst trlst ) 0 1) (diff-size v-acc trlst 1) (len mlst))))

The main decreasing measure is computed by diff-size. It is the number of elements that are in mlst but not in m-acc. A special case is when m is not in mlst, in which case this measure does not decrease. It is not logical to call cs-deadlockfreeₘ if m is not in mlst. Nevertheless, we still need to prove termination for this case, which only occurs on the first call, since get-mlst-route returns messages from mlst. This is why the first element of the measure is 1 if m is not in mlst and 0 otherwise. Similarly, for a logical call of ∨-cs-deadlockfreeₘ, mlst’ is a subset of mlst, but we need to prove termination if this is not the case as well. After the first call the first element decreases to 0 and remains 0. The second element is the main decreasing measure: the difference between m-acc and mlst. This decreases on each call of ∨-cs-deadlockfreeₘ in deadlockfreeₘ and remains equal on each call of deadlockfreeₘ in ∨-deadlockfreeₘ. The third element is 0 for deadlockfreeₘ and 1 for ∨-deadlockfreeₘ. The last element is the self decreasing measure: for deadlockfreeₘ this is constant since it is not self-recursive. For ∨-deadlockfreeₘ this is the length of mlst, since this decreases on each self-call.

We now prove our deadlock prevention theorem, which is an instance of the generic construct given in section 3.3.

(defthm en-route-empty (implies (and (cs-deadlockfree 0 mlst ntkst) (ct-legal-measure measure mlst ns ntkst)) (endp (mv-nth 1 (simple-genoc_t mlst ns measure nil nil time ntkst))))})

The proof follows the methodology from section 3.3. We prove that predicate cs-deadlockfree (1) implies the scheduling assumptions, i.e., implies at least one possible route, and (2) is inductive for GeNoC, i.e., if it holds initially, it holds after one recursion step. We then prove that the measure provided by function Scheduling is always legal. From this follows the theorem above.

The first step consists in proving the theorem below. We assume that there are messages to be sent (line 3) and that each node has *num-of-buffers* buffers (line 4). Furthermore, we assume that the network state relates to the list of messages, i.e., there is no message on the network that is not in mlst. Under these assumptions, predicate cs-deadlockfree is proven to imply a possible route.

(defthm deadlockfree-=>-route-possible 0 (implies (and (consp mlst) (buffersize ntkst *num-of-buffers*) (ntkst-relates-mlst ntkst mlst) (cs-deadlockfree 0 mlst ntkst))) (not (no-good-routes mlst ntkst))))

The second theorem states that cs-deadlockfree is preserved after each recursive call of GeNoC. Its proof consists of proving that if a state is cs-deadlockfree, after a cycle of routing and scheduling the resulting state is still cs-deadlockfree.

The theorem is given below. Let out be the output of function scheduler (line 2). The new network state newntkt and new list of messages newmlst are extracted from out. The new list of messages is routed again (line 5). We prove that if the original state ntkst and the list of messages mlst are cs-deadlockfree, so are the new state and list of messages.

Two assumptions are needed: the network state relates to the list of messages (as explained above), and the routing algorithm will not increase the length of the routes. In our case routes are computed by xy-routing, which is deterministic and minimal. It satisfies the latter assumption.

(defthm genoc-preserves-deadlockfreedom 0 (let* ((out (ct-scheduling mlst ns ntkst)) (newntkt (mv-nth 3 out)) (newmlst (xy-routing (mv-nth 0 out)))) (implies (and (ntkst-relates-mlst ntkst mlst) (mlst-created-by-xy-routing mlst) (deadlockfree 0 mlst ntkst)) (deadlockfree 0 newmlst newntkt))))

As it would result in many complex proofs, we do not attempt to prove the theorem directly. Rather, we prove a more general intermediate lemma. This lemma is based on the notion of two "equally full" network states, formalized by predicate <=-full below. Network state ntkst is <=-full
to network state ntkst2 if (a) ntkst1 is empty, or (b) if the first port of ntkst1 and ntkst2 have both available space and their remaining ports are <=-full.

\[
\text{(defun <=-full (ntkst1 ntkst2)}
\begin{align*}
& \text{(if (endp ntkst1) t} \\
& \quad \text{(let * ((recur (<=-full (cdr ntkst1) (cdr ntkst2)))} \\
& \quad \text{(if (has-empty-buffer (car ntkst2)) (and recur (has-empty-buffer (car ntkst1))) recur)))))
\end{align*}
\]

We now prove our intermediate lemma. Assume a network state ntkst1 and a list m1 of messages. Assume a network state ntkst2 and a list m2 of messages, such that m2 is a sublist of m1 and ntkst2 is <=-full than ntkst1. We now prove that if deadlockfree_m holds for m1 and ntkst1, it holds for m2 and ntkst2.

\[
\text{(defthm abstraction-preserve-deadlockfree_m (implies (and (subsetp newmlst mlst)}
\begin{align*}
& \text{mlst-equal (getcoordinates ntkst) (getcoordinates newntkst))} \\
& \text{((<=-full newntkst ntkst) (deadlockfree_m v m-acc newmlst newntkst)) (deadlockfree_m m m-acc newmlst newntkst))}
\end{align*}
\]

We now prove that the list of delayed and en route messages produced by function ct-scheduler is a sublist of its first input argument, and that the new state produced by this function is <=-full than the current one.

\[
\text{(defthm scheduled-is<=-full-and-subsetp (let* ((out (ct-scheduler mlst ER Arr meas prev ntkst)) (newntkst (mv-nth 3 out)) (newmlst (mv-nth 0 out))) (and ((<=-full newntkst ntkst) (subsetp newmlst mlst)))})
\]

Finally, using the intermediate lemma and the theorem above, we can easily conclude that deadlockfree_m is inductive for GeNoC.

\[
\text{(defthm scheduler-preserves-deadlockfreedom (let* ((out (ct-scheduler mlst ER Arr meas prev ntkst)) (newntkst (mv-nth 3 out)) (newmlst (mv-nth 0 out))) (implies (deadlockfree 0 mlst ntkst)) (deadlockfree 0 newmlst newntkst)))}
\]

4.4 Evacuate!

GeNoC computes two lists: EnRoute and Arrived. Up to this point we proved that EnRoute is empty. Since our injection method injects all messages in the network at the initial simulation step, we can prove that if EnRoute is empty, Arrived is equal to the original list of messages (see Fig. 7).

To prove it we have defined a notion of equivalence for lists of messages, based on the fact that all messages have unique identifiers. Lists m1 and m2 are called mlst-equal if the set of ids of m1 is a subset of the set of the ids of m2, and the other way around. By proving that mlst-equal is an equivalence relation and by proving the following congruences, the degree of automation of ACL2 in proving our theorems significantly increased.

\[
\begin{align*}
& \text{(defequiv mlst-equal)} \\
& \text{(defcong mlst-equal mlst-equal (cons x m) 2)} \\
& \text{(defcong mlst-equal mlst-equal (append m1 m2) 1)} \\
& \text{(defcong mlst-equal mlst-equal (append m1 m2) 2)} \\
& \text{(defcong mlst-equal iff (member-v x m) 2)}
\end{align*}
\]

5. RELATED WORK

Deadlocks can be classified as structural or high-level. Structural deadlocks are often introduced by the routing algorithm. This kind of deadlock has been extensively studied in the context of computer networks [4, 5, 6, 7]. A resource (channels or buffers) dependency graph is constructed using the routing function of the entire network. An acyclic graph is a necessary and sufficient condition for deadlock prevention. This condition is proved under several assumptions. One of them is that when a message reaches its destination, it can always be consumed. This assumption is in practice not satisfied. If a node is really full and cannot process its local input queues, the network might become overloaded and no progress might be possible. Moreover, dependencies between requests and acknowledgment packets may be introduced, creating high-level deadlocks. To prevent such deadlock, data flow analysis is used [11]. Another solution is to include these dependencies in the graphs [16], or to have separate buffers for different types of messages [9, 8].

These techniques need the construction of a graph, and therefore cannot – in contrast to our approach – be applied to parametric models. Moreover, we allow for the unified analysis of the two kinds of deadlocks and their interactions.

6. CONCLUSION

We have presented an extension of the GeNoC model to support the proof that messages injected in a network eventually reach their destination. This has been achieved by defining a generic termination condition – inspired from "clock functions" [12] – and a new proof obligation sufficient to discharge this condition. We also define a general deadlock prevention theorem and instantiated it for a circuit switching technique. In all our proofs, we made no assumption on the topology, the length of messages, or their injection time.

Table 1 gives an overview of the files needed to define and prove the theorems mentioned in this paper. The size of the
The proof that all messages reach their destination is a formal proof of time bounds obtained in such publications.

Acknowledgments
This research is supported by NWO/EW project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.664.811.

7. REFERENCES


