An Improved Fault-Tolerant Routing Algorithm for a Network-on-Chip Derived with Formal Analysis☆

Zhen Zhang[a,*], Wendelin Serwe[b,c,d], Jian Wu[e], Tomohiro Yoneda[f], Hao Zheng[g], Chris Myers[a]

[a] *Dept. of Elec. & Comp. Eng., Univ. of Utah, Salt Lake City, UT, USA*
[b] *Inria*
[c] *Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France*
[d] *CNRS, LIG, F-38000 Grenoble, France*
[e] *Toshiba America Electronic Components, Inc., San Jose, CA, USA*
[f] *National Institute of Informatics, Tokyo, Japan*
[g] *Dept. of Comp. Sci. & Eng., Univ. of S. Florida, Tampa, FL, USA*

Abstract

A fault-tolerant routing algorithm in Network-on-Chip (NoC) architectures provides adaptivity for on-chip communications. Adding fault-tolerance adaptivity to a routing algorithm increases its design complexity and makes it prone to deadlock and other problems if improperly implemented. Formal verification techniques are needed to check the correctness of the design. This paper describes the discovery of a potential livelock problem through formal analysis on an extension of the link-fault tolerant NoC architecture introduced by Wu *et al.* In the process of eliminating this problem, an improved

☆This material is based upon work supported by the National Science Foundation under grants CNS-0930510 and CNS-0930225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Part of this work was performed during a visit of the first author at the Inria Grenoble–Rhône-Alpes research centre. A subset of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

*Corresponding author.

Email addresses: zhen.zhang@utah.edu (Zhen Zhang), Wendelin.Serwe@inria.fr (Wendelin Serwe), bjwujian@gmail.com (Jian Wu), yoneda@nii.ac.jp (Tomohiro Yoneda), zheng@cse.usf.edu (Hao Zheng), myers@ece.utah.edu (Chris Myers)

routing architecture is derived. The improvement simplifies the routing architecture, enabling successful verification using the CADP verification toolbox. The routing algorithm is proven to have several desirable properties including deadlock and livelock freedom, and tolerance to a single-link-fault.

Keywords: fault-tolerant routing, formal methods, model checking, network-on-chip, process calculus

---

1. Introduction

Cyber-physical systems (CPS) have ubiquitous applications in many safety critical areas such as avionics, traffic control, robust medical devices, etc. As an example, the automotive industry makes active use of CPS: modern vehicles can have up to 80 *electronic control units* (ECUs), which control and operate everything from the engine and brakes to door locks and electric windows. Currently, each ECU is statically tied to its specific sensors and actuators. This means that processing power between different ECUs cannot be shared, which degrades the performance of the chip due to imbalanced workload on each ECU. More importantly, this structure is susceptible to faults since if an ECU fails, it causes a malfunction in the corresponding sensor and/or actuator. With advances in semiconductor technology, it is now possible to have multiple cores on a single chip that communicate using the *Network-on-Chip* (NoC) paradigm. A NoC approach allows a flexible mapping of ECUs to processing elements, which makes it possible for ECUs to share processing power and tolerate faults by having spare units.

Wormhole routing has been utilized in many NoC designs, such as Aethereal [1], Hermes [2], and QNoC [3]. It is a switching technique that routes a packet of data in small units, known as *flits*. A packet travels through the network like a worm, and it typically consists of a *header flit* with the packet's destination, *body flits* carrying the packet's information, and a *tail flit* indicating the end of the packet. This paper presents the verification of an asynchronous NoC architecture that supports a link-fault tolerant routing algorithm [4] extended to a multiflit wormhole routing setting. A unique feature of the routing algorithm for this NoC architecture is that it uses a *deadlock avoidance* rather than a *deadlock prevention* scheme. In other words, rather than creating a routing algorithm that is proven to be deadlock-free, potential deadlocks are detected and avoided by dropping packets. To the best of our knowledge, this work is the first formal verification of a dead-

lock avoidance routing function. The verification takes advantage of the CADP (*Construction and Analysis of Distributed Processes*) toolbox [5] and its process-algebraic modeling language LNT (formerly *LOTOS New Technology*) [6].

Previous work on the verification of this routing algorithm [7] applies data abstractions to enable model checking of deadlock freedom and single-link fault tolerance. It, however, is flawed in that the abstract model does not include certain routing failure cases that are present in its concrete counterpart. This paper uses an example to illustrate the flaws in the abstraction and provides a refined, corrected abstraction. During this correction process, redundant fault-tolerance behaviors are found to cause livelocks in the presence of multiple faults. Through a series of diagnostic examples on the concrete model, an improvement to avoid livelocks is made on the NoC architecture and model. Deadlock and livelock freedom, single link-fault tolerance, and packet delivery are formally analyzed. Finally, this paper describes several remaining challenges to the verification of this and similar systems.

This paper is organized as follows. Section 2 describes the NoC architecture and routing algorithm. Section 3 discusses a livelock problem discovered through formal analysis. Section 4 presents an improvement of the NoC routing algorithm to avoid livelocks. Section 5 presents verification results for deadlock and livelock freedom, and properties of single-link fault tolerance and packet delivery. Section 6 surveys related work. Section 7 discusses the insights obtained from using model checking in the design of the NoC architecture and some future research directions.

2. Network-on-Chip Architecture and Routing Algorithm

A fully functional NoC system has to be fault-tolerant and free of deadlocks. The Glass/Ni fault-tolerant routing algorithm [8], guarantees deadlock freedom by disallowing certain turns (i.e., changes in routing direction) in the network, so that communication cycles cannot occur. However, the Glass/Ni routing algorithm uses the *node-fault model*, where a fault in an incoming link is interpreted as the complete node failing. Not only does this mean losing the ability to route to an otherwise functional node, but if the node does not actually stop operating, it can potentially introduce deadlock in the network. Imai *et al.* proposed a modified version [9] that achieves one link-fault tolerance by introducing a mechanism to forward link fault locations to a neighboring routing node allowing for a route selection that avoids the faulty

link. This fault forwarding method though can still result in a deadlock at the edges of a mesh network, so in these cases, it must revert to the node-fault model. Wu *et al.* described an improvement [4] that is capable of handling link faults anywhere in the network. A feature of this routing algorithm is its flexible fault-tolerance mechanism. It handles transient link failures and allows illegal turns whenever there is no danger of a cyclic communication dependency creating a deadlock. In case a potential cyclic communication dependency is detected, deadlock is avoided by dropping the packet that attempts to make an illegal turn, which effectively breaks the cycle. In other words, this routing algorithm only drops packets whenever there is a potential to form a cycle of dependencies. It is this deadlock avoidance scheme that this paper attempts to improve upon and to formally verify.

There are nine types of router nodes for this architecture as depicted in the three-by-three mesh shown in Figure 1. Namely, there are four types of corner nodes (i.e., nodes 00, 02, 20, and 22), four types of edge nodes (i.e., nodes 01, 10, 12, 21), and one type of center node (i.e., node 11). A larger network would include duplicates of the routers shown here. This architecture implements an extended version of the routing algorithm [4] described by Wu *et al*. The original algorithm assumed single-flit packets and that each node could route only a single packet at a time, while this modified architecture allows each node to potentially have several multi-flit packets in flight at a time. For example, node 00 may be routing a packet from node 01 to node 10, while simultaneously routing a packet from node 10 to node 01. To accomplish this, each node xy is composed of several independent *routers* (r_D_xy) and *arbiters* (arb_D_xy), where $D \in \{\mathrm{PE}, \mathrm{E}, \mathrm{N}, \mathrm{S}, \mathrm{W}\}$ corresponds to the direction the packet is coming from in the case of routers and going to in the case of arbiters. Notice that because arbiters need storage capacity to avoid deadlocks [7, Section 4.2], it is necessary to keep them separate from the routers.[1]

The routing algorithm works as follows. Each node communicates with its corresponding *processing element* (PE), and when the PE of a node xy wishes to send a packet to the PE of another node $x'y'$, it injects that packet into

[1]One might argue that it is not necessary to include the arbiters arb_PE_xy in the model, because we assume that a processing element is always ready to consume a packet, so that there is nothing to arbitrate. However, to be closer to the real circuit, we prefer to keep these processes, in particular because they do induce only a small performance penalty in verification execution time.

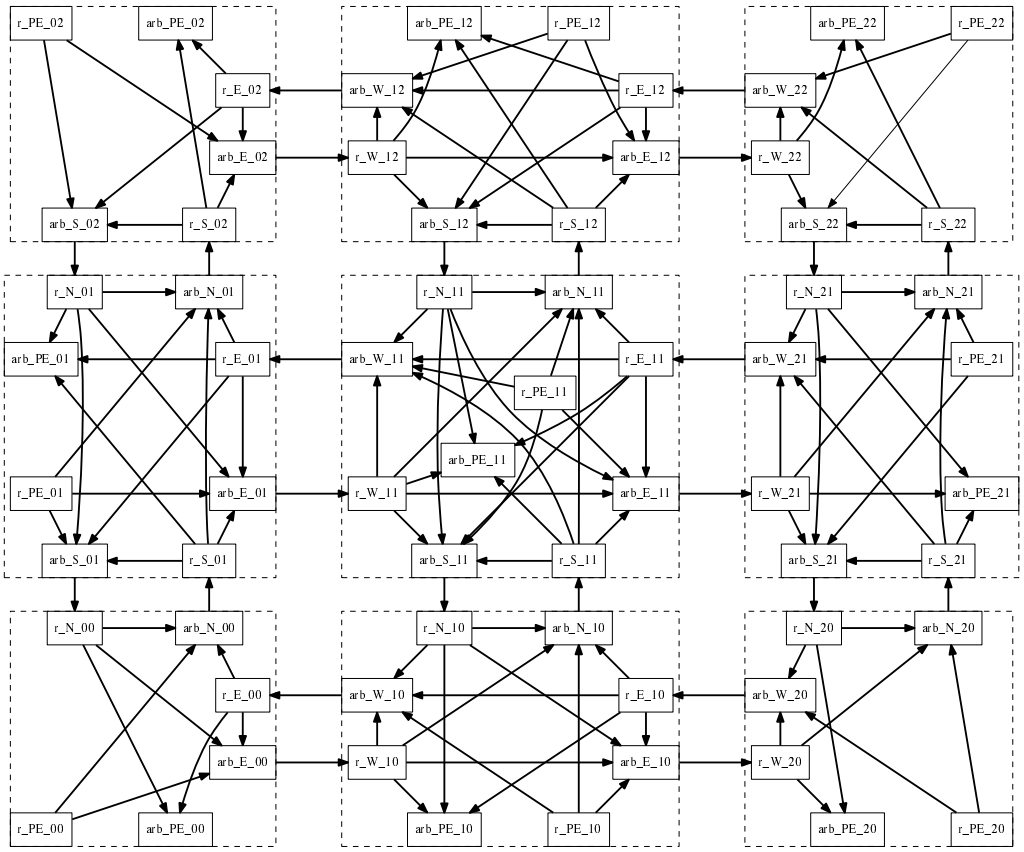

Figure 1: Architecture of the nine routing nodes in a three-by-three mesh.

the network via its *router* (r_PE_xy). Based upon the intended destination of the packet, a router determines a direction D to which to try to forward the packet, and then attempts to communicate with the *arbiter* (arb_D_xy) in charge of the corresponding link. At this point, one of three things can occur. First, the link may be busy, and the router must wait its turn to use the link. Second, the link may be faulty, and the router is instructed to find an alternate route. Finally, the link may be free, and the arbiter may non-deterministically select to communicate with this router over any other routers that may be trying to obtain this link. The arbiter then forwards the packet one flit at a time to the succeeding router (i.e., the router the output

of the arbiter is connected to), which then executes the same algorithm. Once a packet reaches its destination $x'y'$, the packet is consumed by the arbiter connected to its PE (arb_PE_$x'y'$).

To guarantee deadlock freedom, the routing algorithm disallows certain turns in the network. Namely, a packet that is moving north in the network is not allowed to turn to the west, and a packet moving east in the network is not allowed to turn to the south. Hence, in order to avoid illegal turns, each router sends packets south and west, as needed, before sending them north and east. One exception is that if the destination is one node away and to the east (or north) of the source node, i.e. ($x' = x + 1$ and $y' = y$) or ($x' = x$ and $y' = y + 1$), then the r_PE_xy router sends such packets directly east (or north) first.

Assuming there is at most one link-fault, an alternate route always exists, but it may require an illegal turn. For example, consider the two-by-two mesh shown in Figure 2 and assume that node 10 wishes to send a packet to node 01. In this case, a west then north route is the prefered option. If arb_W_10 reports a fault on its link to r_E_00, r_PE_10 must communicate with arb_N_10 instead. Once the packet reaches r_S_11, this router must make an illegal turn and route the packet west through arb_W_11. However, arb_W_11 may be busy routing a packet from node 11 to node 00. This packet in turn may be blocked because arb_S_01 is busy routing a packet from node 01 to node 10. Similarly, this packet may be blocked because arb_E_00 is busy routing a packet from node 00 to node 11. Finally, this packet is blocked because arb_N_10 is busy due to the packet from node 10 to node 01. Taken all together, there is a communication cycle causing a deadlock, as is illustrated in Figure 2. In this case, arb_W_11 sends a *negative acknowledgment* to r_S_11, indicating its unavailability to accept a packet from this router, which tells this router (r_S_11) to drop the incoming packet, removing the communication cycle and avoiding the potential deadlock.

To summarize, the computation of the direction tries to obey the *negative-first* rules proposed by Glass/Ni [8] (rules 1 and 2 below), but when necessary, due to faults, it can make an illegal turn assuming the link is available, otherwise it must drop the packet to avoid deadlock (rule 3):

1. Route the packet west and south to the destination or farther west and south than the destination, avoiding routing the packet to a *negative edge* (i.e., a west or south edge) for as long as possible.

2. Route the packet east and north to the destination, avoiding routing

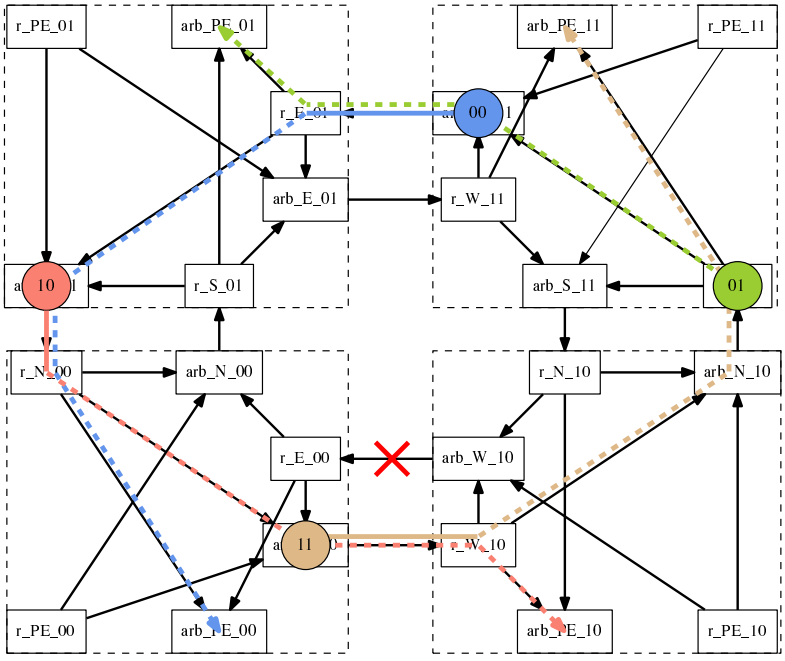

Figure 2: Illustration of a deadlock caused by a cyclic communication dependency. Each packet is represented by a circle containing its destination. A solid line coming out of a packet indicates the router where it is currently traveling to. A dashed line indicates the remaining route of a packet. The crossed link indicates that the link from node 10 to node 00 is faulty.

the packet as far east or north as the destination for as long as possible.

3. If no legal non-faulty link is available, the packet can make an illegal turn (i.e., an east or north going packet may be routed west or south), if the link is available. If the link is busy, the packet must be dropped to avoid deadlock.

Notice that this routing algorithm is not symmetric: Although all routers execute the same algorithm, not all routers behave in exactly the same way, because the routing decision depends not only on the destination of the packet, but also the position of the router.

We modeled the routing architecture in LNT [6], a language for the description of asynchronous concurrent systems and the recommended modeling language for the CADP toolbox (*Construction and Analysis of Distributed Processes*) toolbox [5]. On the one hand, LNT is rooted in concurrency theory, and equipped with a formal semantics in terms of labeled transition systems. On the other hand, to reduce the learning curve, LNT uses a syntax close to those of common programming languages. The complete LNT model is available at `http://www.async.ece.utah.edu/~zhangz/research/lnt_modeling`.

3. Potential Livelock Problem

To analyze the routing protocol, a straightforward approach consists in generating the corresponding *Labeled Transition System* (LTS). If successful, the generated LTS can be used to analyze functional properties of the protocol. The CADP toolbox supports compositional techniques [10] to alleviate the exponential growth of the number of states. In a nutshell, compositional LTS generation proceeds in a "bottom-up" manner, starting with individual processes and alternating generation and minimization steps. CADP features the *Script Verification Language* (SVL) [11], which automates the compositional LTS generation, implementing heuristics [12] to optimize the order processes are taken into account.

Our initial analysis has focused on a two-by-two NoC shown in Figure 3, since state space generation for the three-by-three NoC even using compositional techniques is challenging. The intermediate state space corresponding to only 13 out of the 66 components in the three-by-three NoC already has several hundred million states. Including just one more component to construct the next intermediate LTS almost doubles the size of the LTS. Since

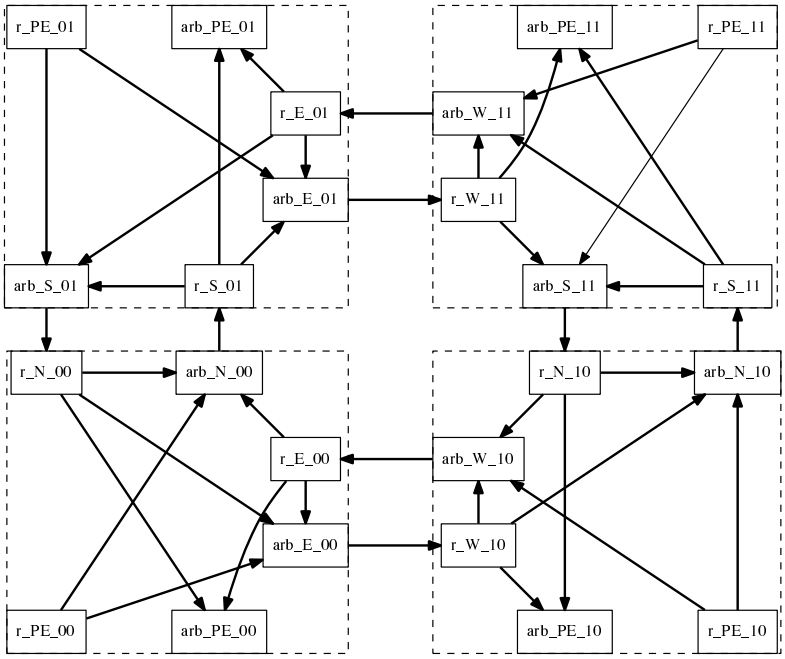

Figure 3: Architecture of the four routing nodes in a two-by-two mesh.

there are still 52 components to be included, it is clear that this *growth* of the intermediate state spaces is unmanageable. Although it might be surprising, these large state spaces can be explained by the fact that each of the 66 components can store a packet (or be empty), resulting in a theoretical state space size of more than 10^{14} states for 14 components of the three-by-three NoC. Although the content of a packet is abstracted to just its destination coordinates, which are necessary to precisely determine a packet's next forwarding direction, the existence of many possible data values further contributes to the combinatorial state explosion.

To alleviate the state explosion problem, a data abstraction method applicable to the network packets is described in our previous work [7]: a packet no longer contains the destination of the packet, and the routing decision steps in the concrete model are replaced by a nondeterministic choice in the abstract model. After receiving a packet, an abstract router nondeterministi-

cally selects either its own node, representing that the packet has reached its destination, or one of the (two, three, or four) forwarding directions, without the need to examine the packets destination. While the exact destination is not needed, it is necessary to keep a Boolean value indicating whether the packet has been diverted, because a router should allow an illegal turn only for packets that have been diverted previously.

Although this data abstraction enables successful verification of properties like deadlock freedom and single-link-fault tolerance, proving packet delivery is impossible, since it is always possible that some, if not all, packets produced get continuously dropped in the network. Using the available information in an abstract packet, one cannot know if a packet reaches its destination or gets dropped by a router on the way. In order to check packet delivery while keeping intermediate state spaces manageable, we investigated a hybrid modeling scheme, combining concrete and abstract packets. The idea is that for one experiment, one node generates a single concrete packet followed by repeated abstract packets while all other nodes only generate abstract packets. In this way, the delivery of a particular concrete packet can be checked with the existence of abstract packets to model network traffic. All routers are modified to handle both types of packets. A router determines the packet's next forwarding direction either precisely based on its destination coordinates or nondeterministically if the packet is abstract. We can then exhaustively run all experiments for every possible concrete packet produced by all four nodes, combined with all possible single fault locations, and check packet delivery properties on the LTS generated for each experiment.

Consider the situation shown in Figure 4 with a faulty link, namely the output of arb_W_10. Router r_PE_10 of node 10 generates a concrete packet destined for node 01. The generated LTS for the concrete model shows that router r_N_10 might fail to find a route for this packet. However, this failure does not exist in the generated LTS for the corresponding abstract model [7]. This mismatch indicates that the abstraction for this router is not correct. The issue is that the fault-tolerant concrete routing logic for r_N_10 provides only one forwarding direction (W) and the only alternative route (N) is forbidden. No routing occurs if the only available route is faulty. However, the corresponding abstract routing behavior still provides the second routing choice, avoiding the routing failure when the first choice is not available. Below, we discuss this problematic example in detail to illustrate the incorrectness of the abstraction and show how it should be fixed.

To send a concrete packet to node 01, r_PE_10 first attempts to send

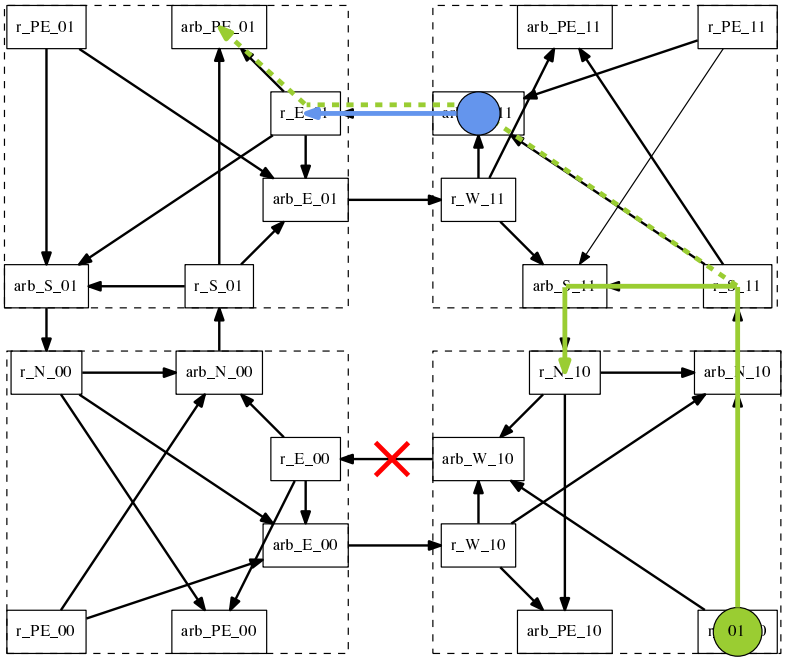

Figure 4: Illustration of the problem with the abstract model. The crossed link indicates that link from node 10 to node 00 is faulty. Solid thick arrows end in the routers handling packets. The dashed arrow indicates the route for the packet from node 10 to node 01, taken into account the failed link.

it west to arb_W_10, but fails because of the faulty link. Then, it diverts the packet to send it north to r_S_11, which then diverts the packet south to r_N_10 because its first preference to the west is blocked (as arb_W_11 is serving another packet). Receiving a concrete packet destined for node 01, r_N_10 of the concrete model only attempts the west direction, and signals a routing failure if the attempt is not successful — r_N_10 does not redirect this packet back to arb_N_10, because it can infer that this packet has been diverted already, based on its location and the packet's destination. Assume a node only generates packets destined for nodes other than itself. So a packet destined for node 01 can only be generated by node 00, 11, or 10. The first choice to forward this packet is north for node 00, and west for

node 11, as either node has two equal choices to send the packet and the first choice is to the shortest path to node 01. For node 10, the preferred choice for this packet is west. Note that none of the preferred choices for node 10 go through r_N_10, which means that if r_N_10 receives a packet for node 10, it must have been diverted.

The LNT description of the abstract model for r_N_10 is shown in Figure 5a. An LNT process has two kinds of parameters: *gates* (between square brackets) and *value parameters* (between parentheses). The latter have the usual meaning, whereas the former define the set of interaction points of the process and yield the labels that may appear on the transitions of the corresponding LTS. LNT uses an Ada-like syntax with usual constructs, such as "**if**−**then**−**else**" or "**loop** ... **end loop**". Variables are declared by a "**var** ... **in** ... **end var**" construct. The LNT construction "**select** A [] B [] C **end select**" is a non-deterministic choice between A, B, and C. Comments start with "--" and extend to the end of the line. For a full description of LNT, its syntax and semantics, the reader is referred to [6].

The router r_N_10 is modeled by the LNT process "*r_N_10_abs*". The first gate parameter "inp" corresponds to the incoming link (connected to the south arbiter of node 11). The next three gate parameters "out_PE", "out_W", and "out_N" correspond to the three externally visible communication links (r_N_10 → arb_PE_10), (r_N_10 → arb_W_10), and (r_N_10 → arb_N_10) in Figure 3, respectively. The fifth gate parameter " fail " does not correspond to a physical communication link but is used for making routing failures visible. The router process uses the variable "pkt" of type Bool to store the (contents of the) packet it forwards. Because the router r_N_10 can never make an illegal turn, the Boolean value contained in a packet is never consulted. Variable "arb_out" of type Bool stores the availability of the arbiter, to which the router is attempting to forward the packet.

The behavior of process "*r_N_10_abs*" is a non-terminating loop consisting of a rendezvous "inp (?pk)", which synchronizes on gate "inp" and stores the received packet in variable "pkt", followed by a non-deterministic choice between sending the packet to its own PE (gate "out_PE", connected to the arbiter arb_PE_10), or forwarding the packet first to the west (gate "out_W", connected to arb_W_10), or to the north (gate "out_N", connected to arb_N_10). Two-way gate rendezvous is used to model packet forwarding: the gate on the router's side can synchronize with the gate on its connected arbiter's side (not shown on the figure). In the non-deterministic choice,

the packet is sent only to a gate that is ready for synchronization; if more than one gate is ready, the choice is non-deterministic; if no gate is ready, the process waits until one of the gates becomes ready. During the rendezvous on a gate, both processes can exchange offers. For example, the "out_PE" gate represents the routers side of the communication link "r_N_10 → arb_PE_10"; hence the value of the packet "pkt" is passed to the receiving arbiter arb_PE_10. A rendezvous on a gate can only happen if both participating processes are ready; otherwise a gate blocks process execution when it waits for synchronization. In the second choice, r_N_10 checks the output link status of its connected arbiter arb_W_10 before sending a packet. This is represented by the rendezvous "out_W(?arb_out)", which waits to receive the status from this arbiter. It sends the packet west through "out_W" if the link is available, otherwise it tries to direct it to the north. If neither choice is feasible, the router performs a rendezvous on gate " fail "; this rendezvous is always possible, because it is not synchronized with any process. This gate rendezvous is referred to as "*route-failure* rendezvous" in the rest of this paper. Execution of a rendezvous on a gate produces a transition labeled with the gate's name and the values of the offer (if any). Notice that the order of the nested "**if**−**then**−**else**" constructs depends on the router, reflecting the asymmetry of the routing function.

The reason why the abstract model does not contain a route failure is that *r_N_10_abs* provides an alternative choice for *any* packet. Moreover, it does not use a packet's diversion status to limit the redundant alternative choice. Consider the case of a packet destined to another node (those destined to 10 can always be routed without failure). Because the other arbiter that r_N_10 connects to, namely arb_N_10, is not faulty in the situation depicted in Figure 4, this means that r_N_10 always sends the packet to arb_N_10 so that the " fail " gate rendezvous never occurs. To refine the abstract model, the router should only divert a packet to an alternative route if it has not been diverted already. Figure 5b shows the added check for the diversion status for the second choice in Figure 5a. A similar correction is made on the third choice which is omitted on Figure 5b.

Although the general principle for the fault-tolerance routing is to provide as much adaptivity as possible, this error in the abstract model illustrates that making multiple diversions can introduce incorrect functional behavior. As described above, r_N_10 can infer the diversion status from a concrete packet destined for node 01 and use it to avoid multiple diversions. On the other hand, if r_N_10 keeps diverting a packet destined for node 01 back to the

```
process r_N_10_abs [inp, out_PE, out_W, out_N, fail: any] is
var pkt: Flit, arb_out: Bool in loop
  inp (?pkt);
  select
     out_PE(pkt)              -- send packet to arb_PE_10
  [] out_W(?arb_out);         -- check arb_W_10's output
     if arb_out then          -- west link to node 00 OK
       out_W(pkt)             -- send packet to arb_W_10
     else                     -- west link to node 00 faulty
       out_N(?arb_out);       -- check arb_N_10's output
       if arb_out then        -- north link to node 11 OK
         out_N(true)          -- send true as packet diverted
       else                   -- north link to node 11 faulty
         fail(pkt)            -- failure to route the packet
     end if end if
  [] out_N(?arb_out);
     if arb_out then
       out_N(pkt)
     else
       out_W(?arb_out);
       if arb_out then out_W(true) else fail(pkt) end if
     end if
  end select
end loop end var end process
```

(a) The original LNT process for abstract r_N_10.

```
out_W(?arb_out);
if arb_out then
  out_W(pkt)
elsif get_diversion(pkt) then
  fail(pkt) -- multiple diversion: route failure
else        -- first diversion: same behavior as above
  out_N(?arb_out);
  if arb_out then out_N(true) else fail(pkt) end if
end if
```

(b) Added check for diversion before sending on alternate route.

Figure 5: The original and modified LNT process for abstract r_N_10.

direction it comes from, it is possible that this packet gets stuck in an infinite *livelock* loop: r_N_10 → arb_N_10 → r_S_11 → arb_S_11 → r_N_10, as shown in Figure 6. Livelock is a scenario where a packet circles around a loop infinitely often without ever reaching its destination. Therefore, avoiding multiple diversions on a NoC is an effective way to prevent livelock. On the fault-free two-by-two NoC in Figure 6, there are only two circular paths, the clockwise inner path and the counterclockwise outer path, that a packet can take to reach any of the four nodes. A router diverts a packet only if it is unable to continue forwarding it on the current circular path due to a link fault. Every time a router diverts a packet, it switches the packet from its current path to the other, effectively reversing its routing direction. The router hopes to deliver the packet through the alternative path. However, if the packet encounters another faulty link on the alternative path, making another diversion puts the packet back to its previous failure path, and the packet is guaranteed to hit the previous faulty link again before it reaches the destination. So having multiple diversions allows a packet to infinitely circle around a loop, which is formed by segments of the two circular paths, i.e., the connected routers and arbiters except for any processing element arbiter (arb_PE_xy).

4. Eliminating Livelock

The existence of livelocks in a NoC routing algorithm can significantly degrade a network's performance, since packets stuck in livelock loops never get delivered, but rather occupy limited bufferring capacities, causing network congestion. Also, repeatedly forwarding packets in livelock loops results in unnecessary power consumption. It is, therefore, important to remove livelocks in our routing algorithm. The simplest solution to eliminate livelocks is to keep track of the diversion status on a packet using an added Boolean variable. This variable, however, requires further space in the packets header (and aggravates the already challenging state explosion problem). It would be better to deduce the diversion status solely from a packet's destination information. This section presents a solution based on this idea through a series of diagnostic examples, which leads to simplifications in both the routing architecture and the routing algorithm.

Clearly, routers making only illegal turns (besides delivering the packet to its destination PE) have superfluous diversions, because in order to make the first illegal turn, the packet must have been diverted already. In a two-

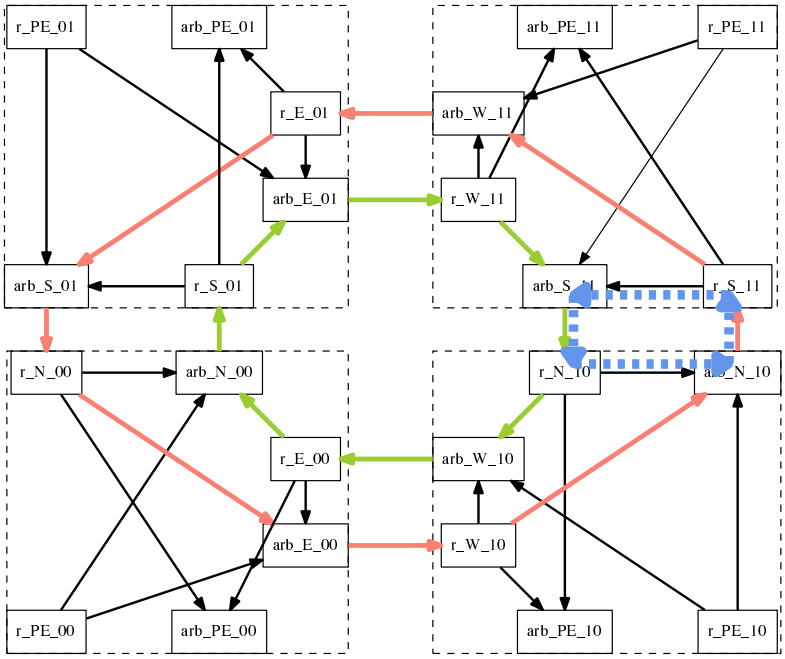

Figure 6: Illustration of the two circular paths that a packet can reach any node and the livelock loop r_N_10 → arb_N_10 → r_S_11 → arb_S_11 → r_N_10.

by-two NoC, for example, on receiving a packet, r_S_11 tries a north-to-west turn first, and tries the north-to-south turn only if the first choice fails. This alternative second choice corresponds to a second diversion and the router should drop the packet if the first route is not available. This also means that r_S_11 does not need to communicate with arb_S_11 and the link between them can be safely removed. This simplification potentially leads to a reduction in the number of gate rendezvous between the two processes during the compositional state space generation. Similarly, the link from r_W_11 to arb_W_11 can be removed.

In general however, inferring the diversion status is difficult as a router does not know a packet's entire forwarding history. As an example, consider all packets that r_N_10 has to forward to its neighboring nodes. It receives all

south-going packets from arb_S_11, and then sends them to either arb_W_10 or arb_N_10. Packets destined for node 11 (arb_PE_11) are not sent through arb_S_11 and hence do not reach r_N_10, and packets destined for node 10 (arb_PE_10) are not forwarded by r_N_10 to its neighbors. This means that packets of interest are destined for either node 01 (arb_PE_01) or the node 00 (arb_PE_00). We know from the previous analysis that all packets destined for node 01 must have been diverted to reach this router. For packets destined for node 00, if they are generated at either node 01 or node 10, then they must have been diverted before reaching r_N_10 as they failed on their preferred choices, i.e., their respective shortest paths to node 00. If a packet is generated at node 11, then its diversion status is not clear as it could be diverted if r_PE_11 forwards it west first or not diverted if south is the first choice. This uncertainty makes it impossible for r_N_10 to infer the diversion status for a packet, since it has no information about the packet's source and its routing preferences.

According to the original description [4] of the routing algorithm by Wu et al, there is no defined order between the two equal routing choices. This means that the implementation of the routing algorithm can bias towards one choice without violating the routing rules. For example, if r_PE_11 always chooses west over south for all packets destined for arb_PE_00, then the said uncertainty at r_N_10 can be resolved. This means that all packets received by r_N_10 are diverted, and this router does not need to divert them again by sending them back north, and the link from r_N_10 to arb_N_10 can be removed. The new LNT process for r_N_10 is shown in Figure 7. After receiving a packet on its "inp" gate that is connected to the output of arb_S_11, it extracts the packet's destination and stores it in "pkt_dest". It then compares the coordinates of the packet's destination with its own location — as usual, the "." notation expresses access to the field of a record. If the destination is reached, it delivers the packet by synchronizing on gate "out_PE" with arb_PE_10. Otherwise, it forwards the packet west if arb_W_10's output link is functional, and fails if it is faulty. A symmetric improvement is made to r_S_01 by tweaking r_PE_00 to make east as its preferred route. This resolves a similar uncertainty that all packets generated by r_PE_00 and destined for arb_PE_11 are routed to the east first. The result is that r_S_01 only receives diverted packets to forward to its east, and the router does not need its output to arb_S_01. This breaks the livelock loop, i.e., arb_N_00→r_S_01→arb_S_01→r_N_00→arb_N_00. Moreover, this modification makes the north-to-south illegal turn disappear, preventing packet drop

```
process r_N_10_concrete [inp, out_PE, out_W, fail: any]
                        (node_loc: Coordinates) is
var pkt: Flit, arb_status: Bool in
   loop
      inp(?pkt);
      if pkt.dest == node_loc then
         out_PE(pkt) -- destination reached
      else -- Only need to try west
         out_W(?arb_status);
         if arb_status then
            out_W(pkt)
         else
            fail(pkt)
         end if
      end if
   end loop
end var end process
```

Figure 7: The new LNT process for r_N_10.

at this router. Note that the assigned ordering between two equal choices is only limited to the mentioned two PE routers when they forward packets destined for nodes in their diagonal directions. Both are equal choices for routing such a packet since each have the same distance to the destination, and therefore no performance penalty is introduced.

Since livelock always occurs on a closed path on the routing architecture, i.e., a path formed by alternating routers and arbiters that enables a packet to circle around indefinitely, it is possible to find livelock by identifying closed paths. For example, a packet destined for arb_PE_11 loops infinitely on the closed path if the output links of arb_N_00 and arb_N_10 are faulty: arb_E_00 $\rightarrow$ r_W_10 $\rightarrow$ arb_W_10 $\rightarrow$ r_E_00 $\rightarrow$ arb_E_00. In this case, r_E_00 can be modified to not divert the packet back east, but only send it north.

The resultant NoC architecture is shown in Figure 8. Note that multiple diversions still exist in some routers. This is because a packet's destination information alone is not sufficient to determine its diversion status at these routers. For example, the packets received by r_E_01 can be diverted, if they are forwarded by r_S_11, and not diverted, if they come from r_PE_11. However, livelock does not occur even if r_E_01 diverts a packet forwarded

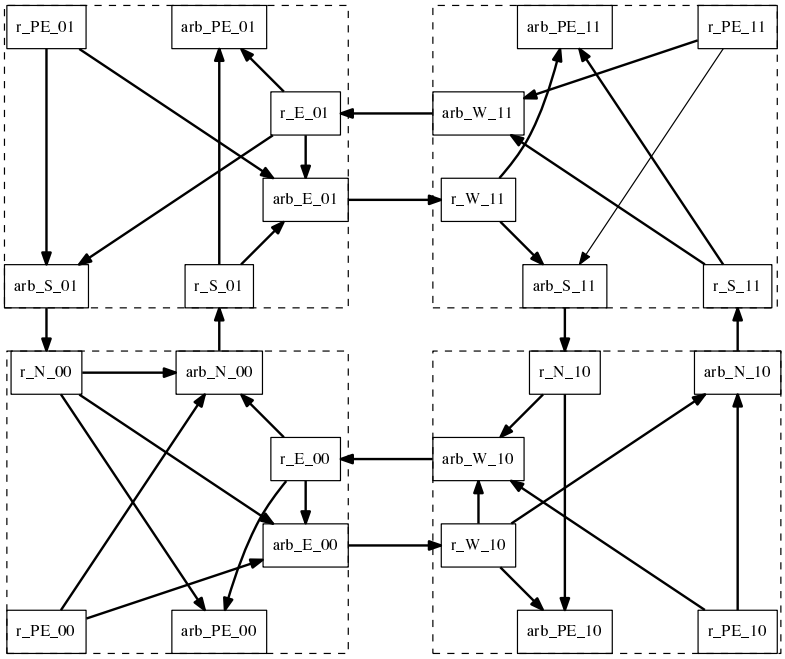

Figure 8: Improved two-by-two NoC architecture with livelock removal. Note that the modifications to avoid livelock have led to the removal of four connections that are no longer needed (i.e., between r_S_01 and arb_S_01, r_W_11 and arb_W_11, r_S_11 and arb_S_11, and r_N_10 and arb_N_10).

by r_S_11 again because this packet is sent to r_N_10 where it gets squashed to avoid multiple diversions. While the improved routing algorithm cannot guarantee that a packet is not diverted more than once, it does guarantee that it is not diverted infinitely often for the two-by-two NoC. This property, however, needs to be formally verified.

Figure 9 presents the decision tree of the livelock free routing protocol in pseudo code. For simplicity, this pseudo code only describes the decision, and it does not provide details on the communication between the routers and arbiters. In particular, determining if a link is faulty requires a communication with the arbiter to determine its current status. This communication is crucial to enable the protocol to adapt to transient failures. Furthermore, in

the case of an illegal turn, the router must determine if the arbiter is busy, and in this case, if it is found to be busy, it drops the packet. Notice that also the modified routing protocol is not symmetric, because the possible routes depend on both the address of the router and the destination of the packet.

5. Verification Results

We applied a two-phase approach to verify a NoC model using the CADP toolbox: first generating the LTS from the LNT specification, and then analyzing the LTS to verify properties of interest. The LTS for each investigated model is generated compositionally [10], i.e., by first generating and minimizing the LTSs for each process separately before alternating combination, hiding, and minimization operations to obtain the LTS of the complete system. For the combination, we applied smart reduction [12], which uses heuristics to find an optimal ordering of the combination and minimization operations to keep the intermediate LTSs manageable. The maximal number of LTSs that can be composed in one step is set to 5. Hiding operations transform into internal transitions those labels that are no longer necessary for further synchronisations or the verification of the properties of interest, greatly enhancing the effectiveness of minimization and verification [13]. To ease the verification tasks, we hide all labels, except those corresponding to route failures and packet drops. Minimization is performed with respect to divergence-sensitive branching bisimulation equivalence [14], a stronger, livelock-preserving variant of branching bisimulation [15].

A desktop machine with a CPU of eight 3.60 GHz cores and 16GB of available RAM is used to generate the results in this section. One core is used at any time for the parallel composition and state minimization steps. One interesting observation is that with the removal of links and simplification of the routing algorithm presented in Section 4, it is possible to completely generate the LTSs of models with concrete packets. All results presented here are based on the two-by-two NoC models with concrete packets. The properties of interest are: (1) the routing algorithm is free of deadlocks and always tolerates a single link fault; (2) it guarantees packet delivery without network traffic; and (3) it is free of livelocks.

5.1. Deadlock Freedom and Single-link-fault Tolerance

In all experiments described in this subsection, each of the four PE routers repeatedly executes the following steps: it first generates a packet with a

- If the packet has reached its destination, deliver the packet.
- else if the packet is one hop away and the corresponding link is available, send on that link
- else
  - go west if:
    1) the current node is not on the west edge,
    2) the packet is going in the west/south direction or just injected,
    3) the west link is fault-free, and
    4) the current node is at or east of the destination OR
       it is at or south of the destination and the south link is faulty
  - else go south if:
    1) the current node is not on the south edge,
    2) the packet is going in the west/south direction or just injected,
    3) the south link is fault-free, and
    4) the current node is at or north of the destination OR
       it is at or west of the destination and the west link is faulty
  - else go east if:
    1) the destination is more than one node to the east OR
       the destination is east of the current node AND exactly one row north,
    2) the packet is not traveling west, and
    3) the east link is fault-free
  - else go north if:
    1) the destination is north of the current node,
    2) the packet is not traveling south, and
    3) the north link is fault-free
  - else
    - go west if:
      1) the current node is not the west edge,
      2) the current node is at or east of the destination,
      3) the packet is not traveling east OR the destination is directly north, and
      4) the west link is fault-free
    - else south if:
      1) the current node is not the south edge,
      2) the current node is at or north of the destination,
      3) the packet is not traveling north, and
      4) the south link is fault-free
    - else go east if:
      1) the current node is at or west of the destination,
      2) the east link is fault-free, and
      3) the packet is not traveling west OR
         the destination is in the current column OR
         the destination is one node to the east AND not one node north
    - else go north if:
      1) the current node is at or south of the destination,
      2) the north link is fault-free, and
      3) the packet is not traveling south OR
         the current node is at or west of the destination

Figure 9: Pseudo-code of the routing protocol.

Table 1: LTSs of the two-by-two NoCs generated for the verification of deadlock freedom and one-fault tolerance.

Failure	Largest Interm. LTS		Final LTS		Performance	
Link	States	Transitions	St.	Tr.	RAM	Time
none	87,040	677,184	1	1	154.7	208.8
$00 \rightarrow 01$	622,080	5,214,528	1	2	224.5	217.7
$00 \rightarrow 10$	469,632	4,252,000	1	2	261.5	221.0
$01 \rightarrow 00$	7,541,100	65,866,878	3	7	7,316.5	910.3
$01 \rightarrow 11$	1620	10,422	1	1	133.6	196.1
$10 \rightarrow 00$	397,575	3,445,506	3	7	133.9	226.7
$10 \rightarrow 11$	2,980,593	27,889,224	1	2	2,752.0	477.6
$11 \rightarrow 01$	1,848	11,830	1	1	138.7	196.0
$11 \rightarrow 10$	52,752	484,572	1	1	139.3	208.2

nondeterministically chosen destination (except to itself), and then sends the packet to its next forwarding direction, which is determined based on the packet's destination. Since each PE router is free to nondeterministically send a packet to any possible destination at any time, our verification generates *all* possible network traffic patterns. It is necessary to model all possible network traffic for the verification of deadlock freedom, since a deadlock can only occur when multiple packets in the network create a communication cycle as described in Section 2, and this cycle is broken by dropping one of the packets. Table 1 shows the LTS information for nine two-by-two mesh models: the first row represents a mesh without any link failure, and the remaining eight rows each represent the same NoC with one failure link whose location is shown in the first column. The columns under "Largest Interm. LTS" show the number of states and transitions of the largest intermediate LTS, and the columns under "Final LTS" show those of the LTS corresponding to the complete model obtained at the end of the compositional generation. The two columns under "Performance" display the maximal amount of allocated virtual memory (in MB) and the total execution time (in seconds) to generate each LTS.

Because the LTS for each model is generated by hiding all gates that represent the links between the routers and the arbiters, the only two visible gates are the route-failure gates and the packet-drop gates. Rendezvous on the former happen when a router has exhausted all options to forward a

packet; rendezvous on the later occurs when a router drops a packet. To model a single link fault in LNT, a working arbiter process is replaced by an arbiter that sends false status to all its connected input routers.

Deadlock freedom is a global property, which requires reasoning about the complete system, and cannot always be inferred from the components taken in isolation. A system has a deadlock if its LTS contains a state/-transition sequence that starts from the initial state and ends in a terminal state, i.e., a state without outgoing transitions. A straightforward approach for deadlock detection is to search for such states in the corresponding LTS. Using the CADP toolbox, it is found that no such sequence exists in any LTS of Table 1. Note that the number of states of the largest intermediate LTS is not proportional to the memory usage. For example, the largest intermediate LTS for the "00 $\rightarrow$ 01" experiment has more states than the one for the "00 $\rightarrow$ 10" experiment, but the latter experiment requires less memory. Similarly, the largest intermediate LTSs in the "01 $\rightarrow$ 11" and "10 $\rightarrow$ 00" experiments are substantially different in size, but their memory usage is comparable. All experiments show that the peak memory usage is a result of minimizing the largest intermediate LTS in each model. Memory usage for state minimization depends, however, not only on the size of the LTS, but also on its branching structure, which explains the weak correlation between the largest LTS and peak memory usage. It is also worth noting that the differences in the state counts clearly show the asymmetry of the routing protocol. Since the entries in Table 1 cover all possible single-link fault configurations, we can conclude that the link-fault tolerant algorithm is free of deadlock for the improved routing algorithm on the two-by-two NoC.

To prove that a router is always able to route a packet, it is necessary to verify that no route-failure gate rendezvous occurs. Since these gates are not hidden during parallel composition, it is straightforward to check their existence in each LTS. Table 2 shows that no transitions are labeled with route-failure labels. This table also shows that with a single failure link in the network, packets may be dropped, namely when the attempt to make an illegal turn could potentially cause a deadlock. The packet drop labels in this table show the location where the drop happens together with the destination of the dropped packet. For example, "drop_Sr_11!Coordinates(0,1)" means a packet destined for node 01 is dropped by the southern router of node 11. The internal transition label is indicated by "i". Therefore, in a highly congested network, dropping packets might happen. Note also that the occurrence of packet drop is more sensitive to certain fault locations than others. Faults

Table 2: Labels of the LTS's corresponding to two-by-two NoCs generated for the verification of deadlock freedom and one-fault tolerance.

Failure	Labels
none	i
$00 \rightarrow 01$	i, drop_Sr_11 !Coordinates (0, 1)
$00 \rightarrow 10$	i, drop_Wr_11 !Coordinates (1, 0)
$01 \rightarrow 00$	i, drop_Wr_11 !Coordinates (1, 0), drop_Wr_11 !Coordinates (0, 0)
$01 \rightarrow 11$	i
$10 \rightarrow 00$	i, drop_Sr_11 !Coordinates (0, 0), drop_Sr_11 !Coordinates (0, 1)
$10 \rightarrow 11$	i, drop_Wr_10 !Coordinates (1, 1)
$11 \rightarrow 01$	i
$11 \rightarrow 10$	i

in one of node 00's two incoming links from node 01 and 10 are responsible for the largest variety of packet drops. But faults in one of node 11's two outgoing links can be entirely tolerated by the routing algorithm without any packet loss.

5.2. Packet Delivery

After the successful verification of deadlock freedom and single-link-fault tolerance, it is important to thoroughly check that each packet can reach its destination. The models of interest for this verification task have at most a single link fault. From the analysis in Section 3, it is known that with two (or more) faulty links, certain routing failures are unavoidable, therefore packet delivery is not guaranteed. Since the routing algorithm guarantees single-link-fault tolerance, it makes sense to check packet delivery on models with at most a single faulty link. Moreover, Table 2 shows that with network traffic, some packets may get dropped instead of reaching their destinations, with even a single link fault. However, packet drop only occurs to avoid deadlock, which requires the existence of network traffic. This means that packet delivery can only be checked on models without any additional network traffic. Therefore, while packet delivery can be verified using the CADP toolbox, in this simple case, it can be ensured by simply confirming that the network remains connected after removing a single failing link.

5.3. Livelock Freedom

In Section 4, a series of diagnostic examples are provided to eliminate livelock issues, which led to simplifications of the NoC architecture. This subsection provides formal analysis of livelock freedom on the simplified NoC. Since it requires at least two faulty links to cause a livelock, with eight external links, there is a total number of $\binom{8}{2} = 28$ different combinations of fault locations. The livelock freedom verification is divided into 28 individual tasks in which each one generates a LTS from a NoC model with a unique pair of link faults.

Similar to the packet delivery verification tasks, only one packet is allowed in the network. A single packet in the network is sufficient for livelock detection, since only a link fault can trigger packet diversion and with multiple diversions a livelock can potentially occur. Traffic may cause a packet to be dropped, but it never causes a packet to be diverted. Therefore, network traffic does not contribute to the cause of livelock. The same configuration for PE routers from Section 5.2 can be used here. As mentioned before, the drawback of this configuration is the introduction of deadlock. Besides, with two link faults, it is unavoidable to have a routing failure that causes deadlock. So, it is certain that some deadlock state exists on the final LTS. However, this fact does not change the results of livelock freedom verification. Livelock is characterized as the existence of an infinite loop on a LTS containing only internal transitions. The goal of checking livelock freedom is to guarantee the absence of such a loop on the final LTS. Thus, the existence of a deadlock state is of no relevance for this verification goal. It is, however, necessary to perform state minimization with respect to divergence-sensitive branching bisimulation equivalence to preserve any actual livelock loop in the NoC model.

For each of the 28 verification tasks, there are 4 different experiments in which one node generates a single packet and then becomes inactive while the other three nodes remain inactive. Gate hiding for each experiment is applied to all communication gates in the model. This means that all previously visible gates are hidden, including the routing-failure gates and packet's generation and consumption gates. Note that gate hiding has the potential danger of turning a cycle into a livelock loop, which causes a false negative result on the final LTS. A cycle differentiates from a livelock loop in that it is a loop with at least one visible transition label representing a communication with its environment. This factor, however, is eliminated in the proposed experiment setting, since with a single packet in the network, there

does not exist meaningful cycles of transitions, such as continuous generation of packets. Hence it does not hamper the detection of real livelocks.

Livelock freedom is verified by checking a simple property requiring the presence of a livelock cycle, which can be expressed in the *Model Checking Language* (MCL) [16] by the following formula:

```
< true* >< "i" > @
```

where "< "i" > @" specifies an infinite loop of internal transitions labeled with "i". The property is satisfied if there exists a state with an outgoing looping internal transition. Violation of this property guarantees that no such state exists and thus shows absence of livelock.

It is found that none of the 112 experiments satisfies the livelock property which proves livelock freedom on the improved routing architecture. From all experiments, the largest intermediate LTS has 112,176 states and 718,564 transitions, and the longest runtime is 175.08 seconds. Each final LTS has a single deadlock state and no transitions. For all experiments, the peak virtual memory used is 114 MB and the total time is about 5.28 hours.

6. Related Work

Besides the Glass/Ni [8] routing algorithm and its link-fault extensions [9, 4], a variety of approaches have been proposed for fault-tolerant NoC routing. A reconfigurable routing table, e.g. [17, 18], is deployed to pre-compute and store routes to avoid faulty links. This method, however, can only avoid permanent faults. Duato analyzes in [19] the effective redundancy for adaptive fault-tolerant routing algorithms, which requires at least four virtual channels per physical channel. However, the use of virtual channels introduces additional area and energy cost. Also, in the case of a single faulty physical link, all virtual links belonging to that faulty physical link become faulty as well. Nordbotten *et al.* [20] use intermediate nodes as backup mechanisms to route packets around network failures. This solution, however, requires extra virtual channels to handle deadlocks, and has to pause all running processes to identify intermediate nodes when a fault is encountered.

Wu presented a fault-tolerant algorithm [21] without the need of using virtual channels. It combines multiple faults to faulty-blocks and routes packets around these faulty-blocks. Similar ideas of binding faulty links and nodes into faulty polygons, chains, and rings have been presented in [22, 23, 24]. Packets are routed around these faulty regions to achieve fault tolerance.

However, a major problem of these approaches is that they create heavy traffic load on the nodes having to route packets around the faulty areas. To obtain a balanced traffic distribution, local or global traffic updates (e.g. [25, 26, 27]) are considered by nodes in the network to route packets.

There have been several previous works that have applied model checking to NoC routing algorithms. For example, to facilitate the use of model checking techniques, automatic translations are developed from the asynchronous hardware description language CHP (*Communicating Hardware Processes*) to networks of automata [28] and to the process-algebraic language LOTOS [29]. The latter approach is applied to verify an input controller [30] for an asynchronous NoC [31] that implements a deadlock-free routing algorithm based on the odd-even turn model [32]. However, this NoC does not handle failures. Deterministic XY routing algorithms, whose routing logic are significantly simpler than the fault-tolerant routing algorithm presented in this paper, have been previously studied [33, 34], leading to the verification of functional properties requiring little network traffic, such as packet delivery. Also, Chen *et al.* face state explosion when attempting to verify deadlock freedom [33], and Palaniveloo and Sowmya mention no results on deadlock verification [34]. Lugan et al. verified an *optical* NoC with four initiators and four targets using Uppaal [35]. To reduce the verification time, their verification used a two-level approach (a first verification on an abstract level, complemented by a more detailed verification of a part of the NoC). Their NoC, however, is not fault-tolerant, and it has highly symmetric processes.

An interesting alternative to model checking is to use static analysis. Verbeek and Schmaltz [36] proposed a necessary and sufficient condition for deadlock-free wormhole routing that can be statically computed independently from the network status. This condition is used in a decision procedure for deadlock detection on large networks from a wide range of NoC topologies and routing algorithms [37]. With the help of the DCI2 (*Deadlock Checker In Designs of Communication Interconnects*) tool [38], Alhussien *et al.* [39] proved deadlock-freeness, livelock-freeness and packet delivery of a fault-tolerant wormhole routing logic for large scale mesh networks. A formal NoC specification and validation environment, GeNoC (*Generic Network-on-Chip*), implemented in the ACL2 theorem prover was first proposed by Borrione *et al.* [40]. It was used to verify a non-minimal adaptive routing algorithm in [41]. Recently, an improvement of the GeNoC model [42] is proposed to enable static verification of deadlock freedom and livelock freedom, as well as functional correctness. It was shown to prove these prop-

erties on an adaptive west-first routing algorithm on a Hermes NoC, with approximately 86 percent of the proof automatically derived. By using static analysis, both the DCI2 and GeNoC approaches are extremely efficient for checking the same properties checked in this paper for deadlock prevention routing algorithms. This efficiency enables them to be applied to large networks. These approaches, however, are not capable of verifying properties of deadlock avoidance algorithms as this requires a dynamic analysis.

To better understand the value of static analysis, we attempted to encode our NoC routing algorithm using the DCI2 approach. Using DCI2, the routing algorithm is encoded as a function in the C language. This function determines the next direction to route based on the current state (i.e., current node, destination of packet, and where the packet came from). While the LNT description includes implementation details, such as the connections between the routers and arbiters and their interaction between routers and arbiters to check for availability, the DCI2 approach abstracts away these details. So, while DCI2 can verify a static model of the protocol, it does not verify the implementation architecture for the protocol nor its dynamic behavior. Using DCI2, it is possible to show that the routing protocol without faults is deadlock-free and livelock-free for larger networks (we checked up to five-by-five router nodes). It is also possible to verify that there are no disconnected routes in the presence of a single fault. In the single fault case though, DCI2 reports deadlocks, since there is no mechanism to encode packet dropping to avoid deadlock in DCI2. Finally, in the double fault case, DCI2 reports both deadlocks and disconnected routes though no livelocks are found. While this is reassuring, these results should be confirmed with model checking, because our deadlock avoidance routing protocol cannot be precisely encoded in DCI2.

7. Discussion

In order to enable the verification of packet delivery, this paper first presents a hybrid modeling scheme as an extension to the previously presented pure abstract model [7]. The discovery of routing failure hidden by the abstract model for router r_N_10 leads to the detection of the potential danger of multiple packet diversions in the original routing algorithm. It is found that excessive fault-tolerance in terms of multiple packet diversions not only fails to increase the chance of delivering a packet, but also potentially

causes livelock problems where a packet circles around an infinite loop and never reaches its destination.

Multiple packet diversions are then analyzed in detail through a series of diagnostic examples on the concrete NoC model. The routing algorithm is corrected and certain routers are restricted to avoid multiple diversions and eliminate potential livelock loops. Since the diversion status is not directly encoded in a concrete packet, it is not always possible to infer from the packet's destination information. Therefore, biased choice between two equally preferred routes is assigned to some PE routers, so that only diverted packets are received by those routers that cannot always decide upon the diversion status of packets. These modifications make it possible to remove redundant communication links on the routing architecture. This simplification eventually leads to manageable state space generation of the concrete model for the two-by-two NoC.

With the help of the CADP verification toolbox, several interesting functional behaviors of the improved routing algorithm are analyzed. Deadlock freedom and single-link-fault tolerance are proved in a congested network with zero or one link fault. Under the single-link-fault condition, a packet is successfully delivered to its destination or dropped due to deadlock avoidance. Without the network traffic, packet delivery is guaranteed. The absence of livelock is proved under two link-fault conditions.

Experience gained in livelock elimination for the two-by-two case provides us valuable insights to determine the appropriate fault tolerance in the routing algorithm. These insights can guide the design of more complex routing behaviors in a large network, as well as abstractions of these complex designs. Formal analysis of these complex designs is necessary to guarantee their functional correctness. Also, diagnostic counterexamples generated from property checking may potentially be useful to refine a model's abstraction.

While model checking is needed to check implementation details, such as connectivity of routers and arbiters, and dynamic properties, such as deadlock avoidance, static analysis methods such as those used by DCI2 and GeNoC can be much more efficient. Therefore, an interesting area of future research is to develop methodologies that leverage both techniques. For example, DCI2 can be used to refine a routing protocol to eliminate deadlocks, livelocks, and disconnected route conditions. Once the routing protocol is developed, model checking can be employed to check the implementation architecture and verify dynamic properties.

Table 2 indicates that packet drop can limit the effectiveness of the routing

algorithm, especially when a particular link is faulty. Packet drop, however, is necessary for deadlock avoidance in our link-fault routing algorithm. It is a challenging task to justify the performance of the routing algorithm in terms of packet drop due to deadlock avoidance in a pure functional verification setting. To obtain a more accurate justification, it seems promising to annotate transitions on the LTS obtained for the functional analysis with link failure probabilities, and then apply quantitative methods such as Markov Chain analysis to evaluate its performance.

[1] K. Goossens, J. Dielissen, A. Rădulescu, ÆThereal network on chip: Concepts, architectures, and implementations, IEEE Design & Test of Computers 22 (5) (2005) 414–421. doi:10.1109/MDT.2005.99.

[2] F. Moraes, N. Calazans, A. Mello, L. Möller, L. Ost, Hermes: An infrastructure for low area overhead packet-switching networks on chip, Integration, the VLSI Journal 38 (1) (2004) 69–93. doi:10.1016/j.vlsi.2004.03.003.

[3] E. Bolotin, I. Cidon, R. Ginosar, A. Kolodny, Qnoc: Qos architecture and design process for network on chip, Journal of Systems Architure 50 (2-3) (2004) 105–128. doi:10.1016/j.sysarc.2003.07.004.

[4] J. Wu, Z. Zhang, C. Myers, A fault-tolerant routing algorithm for a network-on-chip using a link fault model, in: Virtual Worldwide Forum for PhD Researchers in Electronic Design Automation, 2011.

[5] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: a toolbox for the construction and analysis of distributed processes, Springer International Journal on Software Tools for Technology Transfer (STTT) 15 (2) (2013) 89–107. doi:10.1007/s10009-012-0244-z.

[6] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, G. Smeding, Reference manual of the LNT to LOTOS translator (version 6.3), INRIA/VASY/CONVECS (Nov. 2015).

[7] Z. Zhang, W. Serwe, J. Wu, T. Yoneda, H. Zheng, C. Myers, Formal analysis of a fault-tolerant routing algorithm for a network-on-chip, in: F. Lang, F. Flammini (Eds.), Formal Methods for Industrial Critical Systems, Vol. 8718 of Lecture Notes in Computer Sci-

ence, Springer International Publishing, 2014, pp. 48–62. doi:10.1007/978-3-319-10702-8_4.

[8] C. J. Glass, L. M. Ni, Fault-tolerant wormhole routing in meshes, in: Digest of Papers of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing FTCS-23 (Toulouse, France), IEEE Computer Society, 1993, pp. 240–249. doi:10.1109/FTCS.1993.627327.

[9] M. Imai, T. Yoneda, Improving dependability and performance of fully asynchronous on-chip networks, in: Proceedings of the 2011 17th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 65–76. doi:10.1109/ASYNC.2011.15.

[10] H. Garavel, F. Lang, R. Mateescu, Compositional Verification of Asynchronous Concurrent Systems using CADP, Acta Informatica 52 (4) (2015) 337–392. doi:10.1007/s00236-015-0226-1.

[11] H. Garavel, F. Lang, SVL: a Scripting Language for Compositional Verification, in: Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001, Kluwer Academic Publishers, 2001, pp. 377–392, full version available as INRIA Research Report RR-4223. doi:10.1007/0-306-47003-9_24.

[12] P. Crouzen, F. Lang, Smart Reduction, in: D. Giannakopoulou, F. Orejas (Eds.), Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering FASE 2011 (Saarbrücken, Germany), Vol. 6603 of Lecture Notes in Computer Science, Springer Verlag, 2011, pp. 111–126. doi:10.1007/978-3-642-19811-3_9.

[13] R. Mateescu, A. Wijs, Property-Dependent Reductions for the Modal Mu-Calculus, in: A. Groce, M. Musuvathi (Eds.), Proceedings of the 18th International SPIN Workshop on Model Checking Software SPIN'2011 (Snowbird, UT, USA), Vol. 6823 of Lecture Notes in Computer Science, Springer Verlag, 2011, pp. 2–19. doi:10.1007/978-3-642-22306-8_2.

[14] R. J. van Glabbeek, B. Luttik, N. Trcka, Branching bisimilarity with explicit divergence, Fundamenta Informaticæ 93 (4) (2009) 371–392.

[15] R. J. van Glabbeek, W. P. Weijland, Branching-Time and Abstraction in Bisimulation Semantics (extended abstract), CS R8911, Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands, also in Proceedings of the 11th IFIP World Computer Congress, San Francisco, 1989 (1989).

[16] R. Mateescu, D. Thivolle, A model checking language for concurrent value-passing systems, in: Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland), Vol. 5014 of Lecture Notes in Computer Science, Springer Verlag, 2008, pp. 148–164. doi:10.1007/978-3-540-68237-0_12.

[17] R. Casado, A. Bermúdez, F. J. Quiles, J. L. Sánchez, J. Duato, A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks, IEEE Transactions on Parallel and Distributed Systems 12 (2) (2001) 115–132. doi:10.1109/71.910868.

[18] D. Fick, A. DeOrio, G. Chen, V. Bertacco, D. Sylvester, D. Blaauw, A Highly Resilient Routing Algorithm for Fault-tolerant NoCs, in: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, 2009, pp. 21–26.

[19] J. Duato, A theory of fault-tolerant routing in wormhole networks, IEEE Transactions on Parallel and Distributed Systems 8 (8) (1997) 790–802. doi:10.1109/71.605766.

[20] N. A. Nordbotten, M. E. Gómez, J. Flich, P. López, A. Robles, T. Skeie, O. Lysne, J. Duato, A fully adaptive fault-tolerant routing methodology based on intermediate nodes, in: Network and Parallel Computing, IFIP International Conference, NPC 2004, Wuhan, China, October 18-20, 2004, Proceedings, 2004, pp. 341–356. doi:10.1007/978-3-540-30141-7_49.

[21] J. Wu, A fault-tolerant and deadlock-free routing protocol in 2d meshes based on odd-even turn model, IEEE Transactions on Computers 52 (9) (2003) 1154–1169. doi:10.1109/TC.2003.1228511.

[22] R. Boppana, S. Chalasani, Fault-tolerant wormhole routing algorithms for mesh networks, IEEE Transactions on Computers 44 (7) (1995) 848–864. doi:10.1109/12.392844.

[23] K. Chen, G. Chiu, Fault-tolerant routing algorithm for meshes without using virtual channels, J. Inf. Sci. Eng. 14 (4) (1998) 765–783.

[24] J. Zhou, F. C. M. Lau, Adaptive fault-tolerant wormhole routing in 2d meshes, in: Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, April 23-27, 2001, 2001, p. 56. doi:10.1109/IPDPS.2001.925000.

[25] T. T. Ye, L. Benini, G. De Micheli, Packetization and routing analysis of on-chip multiprocessor networks, Journal of Systems Architecture 50 (2-3) (2004) 81–104. doi:10.1016/j.sysarc.2003.07.005.

[26] I.-G. Lee, J. Lee, S.-C. Park, Adaptive routing scheme for NoC communication architecture, in: Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on, Vol. 2, 2005, pp. 1180–1184. doi:10.1109/ICACT.2005.246172.

[27] T. Schonwald, J. Zimmermann, O. Bringmann, W. Rosenstiel, Fully adaptive fault-tolerant routing algorithm for network-on-chip architectures, in: Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on, 2007, pp. 527–534. doi:10.1109/DSD.2007.4341518.

[28] D. Borrione, M. Boubekeur, L. Mounier, M. Renaudin, A. Sirianni, Validation of asynchronous circuit specifications using IF/CADP, in: VLSI-SOC: From Systems to Chips, Selected papers from the 12th IFIP International Conference on VLSI, Vol. 200, International Federation for Information Processing, 2006, pp. 85–100. doi:10.1007/0-387-33403-3_6.

[29] H. Garavel, G. Salaün, W. Serwe, On the semantics of communicating hardware processes and their translation into LOTOS for the verification of asynchronous circuits with CADP, Sci. Comput. Program. 74 (3) (2009) 100–127. doi:10.1016/j.scico.2008.09.011.

[30] G. Salaün, W. Serwe, Y. Thonnart, P. Vivet, Formal verification of CHP specifications with CADP illustration on an asynchronous Network-on-Chip, in: Asynchronous Circuits and Systems, 2007. ASYNC 2007. 13th IEEE International Symposium on, 2007, pp. 73–82. doi:10.1109/ASYNC.2007.18.

[31] E. Beigné, F. Clermidy, P. Vivet, A. Clouard, M. Renaudin, An Asynchronous NOC Architecture Providing Low Latency Service and Its Multi-Level Design Framework, in: Proceedings of the 11th International Symposium on Advanced Research in Asynchronous Circuits and Systems ASYNC 2005 (New York, USA), IEEE Computer Society, 2005, pp. 54–63. doi:doi:10.1109/ASYNC.2005.10.

[32] G.-M. Chiu, The odd-even turn model for adaptive routing, IEEE Transaction on Parallel and Distributed Systems 11 (7) (2000) 729–738. doi:10.1109/71.877831.

[33] Y.-R. Chen, W.-T. Su, P.-A. Hsiung, Y.-C. Lan, Y.-H. Hu, S.-J. Chen, Formal modeling and verification for network-on-chip, in: Green Circuits and Systems (ICGCS), 2010 International Conference on, 2010, pp. 299–304. doi:10.1109/ICGCS.2010.5543050.

[34] V. A. Palaniveloo, A. Sowmya, Application of formal methods for system-level verification of network on chip, in: IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2011, 4-6 July 2011, Chennai, India, 2011, pp. 162–169. doi:10.1109/ISVLSI.2011.57.

[35] L. G. Iugan, G. Nicolescu, I. O'Connor, Modeling and formal verification of a passive optical network on chip behavior, Electronic Communications of the EASST 21. doi:10.14279/tuj.eceasst.21.302.

[36] F. Verbeek, J. Schmaltz, On necessary and sufficient conditions for deadlock-free routing in wormhole networks, IEEE Transactions on Parallel and Distributed Systems 22 (12) (2011) 2022–2032. doi:10.1109/TPDS.2011.60.

[37] F. Verbeek, J. Schmaltz, A decision procedure for deadlock-free routing in wormhole networks, IEEE Transactions on Parallel and Distributed Systems 25 (8) (2014) 1935–1944. doi:10.1109/TPDS.2013.121.

[38] F. Verbeek, J. Schmaltz, Automatic verification for deadlock in networks-on-chips with adaptive routing and wormhole switching, in: NOCS 2011, Fifth ACM/IEEE International Symposium on Networks-on-Chip, Pittsburgh, Pennsylvania, USA, May 1-4, 2011, 2011, pp. 25–32.

[39] A. Alhussien, F. Verbeek, B. van Gastel, N. Bagherzadeh, J. Schmaltz, Fully reliable dynamic routing logic for a fault-tolerant NoC architecture, Journal of Integrated Circuits and Systems 8 (1) (2013) 43–53.

[40] D. Borrione, A. Helmy, L. Pierre, J. Schmaltz, A formal approach to the verification of networks on chip, EURASIP Journal Embedded Systems 2009 (2009) 2:1–2:14. doi:10.1155/2009/548324.

[41] A. Helmy, L. Pierre, A. Jantsch, Theorem proving techniques for the formal verification of NoC communications with non-minimal adaptive routing, in: Proceedings of the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS 2010 (Vienna, Austria), 2010, pp. 221–224. doi:10.1109/DDECS.2010.5491781.

[42] F. Verbeek, J. Schmaltz, Easy formal specification and validation of unbounded networks-on-chips architectures, ACM Transactions on Design Automation of Electronic Systems 17 (1) (2012) 1:1–1:28. doi:10.1145/2071356.2071357.