Scaling Up Livelock Verification for Network-on-Chip Routing Algorithms

Landon Taylor(✉) and Zhen Zhang

Utah State University, Logan, UT, USA
{landon.jeffrey.taylor,zhen.zhang}@usu.edu

Abstract. As an efficient interconnection network, Network-on-Chip (NoC) provides significant flexibility for increasingly prevalent many-core systems. It is desirable to deploy fault-tolerance in a dependable safety-critical NoC design. However, this process can easily introduce deeply buried flaws that traditional simulation-based NoC design approaches may miss. This paper presents a case study on applying scalable formal verification that detects, corrects, and proves livelock in a dependable fault-tolerant NoC using the IVy verification tool. We formally verify correctness at the routing algorithm level. We first present livelock verification using refutation-based simulation scaled to a 15-by-15 two-dimensional NoC. We then present a novel zone-based approach to livelock verification in which finite coordinate-based routing conditions are abstracted as positional zones relative to a packet's destination. This abstraction allows us to detect and remove livelock patterns on an arbitrarily large network. The resultant improved routing algorithm is free of livelock and maintains a high level of fault tolerance.

Keywords: Network-on-Chip · Fault-tolerant routing · Model checking · Property-directed reachability

1 Introduction

Network-on-Chip (NoC) is an interconnection network that governs on-chip communication among homogeneous routers for many-core systems. NoC provides flexibility in balancing processing load among interconnected cores to optimize power and tolerate faulty connections. As computing systems advance, many-core systems increase design complexity, and NoC provides an efficient solution to this challenge [17,21]. When NoC is used in safety-critical applications such as electronic control units in a vehicle [26], it must provide provable correctness guarantees. A dependable NoC routing algorithm must tolerate faulty links to minimize the impact on processing cores. Fault tolerance improves network dependability by allowing a network to route otherwise blocked packets to their destinations. However, the complexity of fault-tolerant routing design is liable

B. Finkbeiner and T. Wies (Eds.): VMCAI 2022, LNCS 13182, pp. 378–399, 2022.
https://doi.org/10.1007/978-3-030-94583-1_19

to include flaws that traditional simulation and testing methods may miss. A flaw in a routing algorithm may produce livelock, which results in packets traveling cyclically forever while wasting power and worsening network traffic. Formal verification of livelock freedom on NoC designs remains challenging today.

This paper presents a case study on scaling formal verification of a complex fault-tolerant routing algorithm designed to operate on either a synchronous or an asynchronous NoC routing architecture. We rebut the livelock-freedom claim made in [29] by finding livelock traces in the same routing algorithm. We then demonstrate significantly improved scalability of livelock freedom verification on arbitrarily large two-dimensional mesh networks. The paper then presents the proven livelock-free routing algorithm.

Our approach evaluates the high-level routing algorithm using the IVy tool [19]. We describe our verification approach as follows:

- We first simulate the routing behavioral model to prove packet delivery and prove that any discovered potential livelock traces indicate true livelock scenarios. We use this approach to verify livelock freedom in NoCs of size 3×3 to 15×15.
- Next, we describe a highly automated method to fix the routing algorithm by incrementally removing livelock traces. Eventually, this produces a livelock-free and fault-tolerant routing algorithm.
- To further scale up livelock verification, we present an incremental abstraction approach to derive routing zones on an arbitrary $m\times n$ network, followed by the derivation of abstract moves to allow efficient representation of very large livelock patterns.

This paper is organized as follows. Sections 2 and 3 describe background and related work, respectively. Section 4 introduces the link-fault routing algorithm analyzed in this paper. Sections 5, 6, and 7 present our refutation-based simulation approach for livelock checking and livelock removal. Sections 8, 9, and 10 present our zone-based routing model and livelock verification scaled to arbitrarily large network. Section 11 concludes the work. Data, supplemental material, and models from this work can be found on GitHub[1].

2 Preliminaries

Inductive Invariant Verification. The IVy tool supports interactive inductive invariant strengthening and verification [18]. Using the Z3 SMT solver [20], IVy can interactively aid a user to strengthen invariants. It starts by checking the user-provided invariant for inductiveness and returns a counterexample to induction if the invariant fails to be inductive. It then guides the user with recommendations to strengthen the invariant. Once the user strengthens the invariant, it checks for inductiveness again. These counterexamples to induction and invariant-strengthening recommendations prove invaluable to our work by providing traces of livelock scenarios. A recent addition to IVy that has

[1] https://github.com/formal-verification-research/IVy-Models.

proved pivotal to this research is integration with Property-Directed Reachability (PDR) in the ABC model checking tool [2,6]. PDR Automatically strengthens invariants to use inductiveness checking as reachability verification.

Network-On-Chip and Livelock. In this paper, we use the terms "NoC" and "network" interchangeably. We consider a two-dimensional NoC in a square mesh and model it as a coordinate system composed of $n \times n$ nodes ($n \geqslant 2$ and $n \in \mathbb{N}$). A *node* (x_i, y_i) is represented as a coordinate pair identified by an index i. The subscript i has no relation to the location of the packet in the network; rather, it represents the number of times the packet has been forwarded to reach a node. For instance (x_4, y_4) is the fourth node a packet visits, and its coordinates may be $(3, 3)$, and (x_5, y_5) may have coordinates $(3, 2)$. Nodes exchange information by sending each other packets. In this work, a *packet* is assumed to only carry its destination coordinate (x_d, y_d). Packets travel through a network following a pre-defined routing algorithm. We present the formal analysis and correction of an adaptive routing algorithm that tolerates faults *dynamically*, i.e., the routing algorithm does not know fault locations in the network and it selects an alternative route for each packet whenever it encounters a fault on its way to the destination. Therefore, a packet's route from its source to destination is not statically determined beforehand. Each node in the network is composed of *routers* that determine a packet's next forwarding direction based on its intended destination and *arbiters* that resolve simultaneous packet forwarding requests to compete for the output channel in the same direction.

As a packet travels through the network, it produces a *trace*, which records the history of visited nodes. A trace begins at (x_0, y_0) where the packet is generated and is represented by $(x_0, y_0), (x_1, y_1), \ldots, (x_i, y_i)$. Define a *livelock pattern* as (x_i, y_i), (x_{i+1}, y_{i+1}), $\ldots$, $(x_{i+k}, y_{i+k}), (x_i, y_i)$, (x_{i+1}, y_{i+1}), $\ldots$, (x_{i+k}, y_{i+k}), $\ldots$, where $1 \leq k \leq K$ and $K \in \mathbb{N}$. Nodes (x_i, y_i) and (x_{i+K}, y_{i+K}) are the first and the final nodes in the sequence of repeated nodes constituting the livelock pattern, respectively. A vital part of a livelock pattern is cyclical behavior. That is, a livelock pattern includes a series of repeated changes in traveling direction. For instance, a livelock pattern starting at (x_2, y_2) consisting of a packet traveling back and forth between two nodes can be represented by $(x_2, y_2), (x_3, y_3), (x_2, y_2), (x_3, y_3), \ldots$, which may be shown using coordinates as $(2, 1), (2, 2), (2, 1), (2, 2), \ldots$. A *livelock prefix* is a finite trace represented by $(x_0, y_0), (x_1, y_1), \ldots, (x_{i-1}, y_{i-1})$ and (x_i, y_i) is the first node in a livelock pattern. A prefix may be empty in the case where $i = 0$. For instance, a prefix for the pattern above with $(x_i, y_i) = (2, 1)$ may be $(x_0, y_0), (x_1, y_1)$, or $(1, 0), (1, 1)$. A *livelock trace* consists of a livelock prefix followed *immediately* by a livelock pattern. For example, combining the prefix and pattern listed above produces the trace $(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_2, y_2), (x_3, y_3), \ldots$, which may be represented using coordinates as $(1, 0), (1, 1), (2, 1), (2, 2), (2, 1), \ldots$. A *livelock-free trace* does not include a livelock pattern. If the set of all traces produced in a network contains only livelock-free traces, the network is a *livelock-free network*. In other words, a livelock-free network is one in which no packet can generate a livelock trace.

3 Related Work

Formal verification techniques have been applied to safe and reliable NoC designs at different levels of abstraction. An overview of recent work can be found in [4]. Dridi et al. [10] modeled a double arbiter and a switching router in the IF language [5] and verified circuit-level safety properties. Zaman et al. [27] verified functional correctness on networks up to 8×8 using the SPIN model checker [14]. They randomly simulated the NoC model to eliminate property violation scenarios before applying model checking. Similarly, we find that simulation enables rapid discovery of potential livelock cases that can be further proved by formal techniques. Van Gastel et al. [12] used the xMAS language [8] to formally define executable specifications of micro-architectures. Using the ABS language [16], Din et al. [9] verified livelock freedom using invariants to monitor their local history. In comparison, our proposed livelock freedom verification work checks stronger properties, including termination of each packet's travel. Particularly important to NoC dependability is a fault-tolerant routing algorithm. Imai et al. [15] proposed a link-fault location forwarding mechanism to achieve single link-fault tolerance. Zhang et al. [28,29] modeled an improved link-fault-tolerant routing algorithm [25] in the process-algebraic language LNT [7] and proved deadlock- and livelock-freedom, as well as, tolerance to a single-link fault using the CADP toolbox [11] for 2×2 NoCs. These approaches, however, encountered significant challenges in scaling the verification to larger NoCs.

In addition to model checking, theorem proving has been applied to NoC verification. The Generic Network-on-Chip [3] framework was created with the help of the ACL2 theorem prover and was used to verify non-minimal adaptive routing algorithms in [13]. Verbeek et al. [24] proved livelock- and deadlock-freedom for an adaptive west-first routing algorithm on a Hermes NoC, with approximately 86% of the proof automatically derived. The prototype tool DCI2 (*Deadlock Checker In Designs of Communication Interconnects*) [23] implements necessary and sufficient conditions for deadlock-free routing and was used for deadlock detection in a range of NoCs [22]. This tool requires a user to define a network topology, size, and routing algorithm [1]. Using this tool, Zhang et al. [29] verified livelock freedom for up to 5×5 NoCs for the link-fault-tolerant routing algorithm in [25]. However, this tool is impacted by combinatorial blow-up when the size of a NoC is increased.

Work presented in this paper drastically scales formal verification of the link-fault-tolerant algorithm in [29] to arbitrarily large NoCs. While [29] illustrates limitations of tools like DCI2 [23] and LNT [7], this work describes a novel and effective way to scale-up livelock verification. We show that property-directed reachability using the IVy verification tool is an efficient way to verify complicated NoC routing algorithms, especially compared to enumerative model checking approaches. Compared to similar recent work, this work places an emphasis on guiding a user through abstracting and improving the routing algorithm for a fault-tolerant network. Moreover, we rebut the livelock freedom proof in [29] by showing the existence of livelock traces on a 3×3 NoC which DCI2 did not detect according to [29]. We derive and prove a correct livelock-free routing algorithm.

4 Link-Fault-Tolerant Routing Algorithm

Originally presented in Fig. 9 of [29], this algorithm is adapted as Algorithm 1 with variables defined in Table 1. It operates on an $m \times n$ mesh network with no virtual channels. Figure 1 provides an example of routing decisions according to Algorithm 1. Unless the packet is at the destination or can make one hop to reach the destination, it is first routed west and south towards the destination. A packet is routed west even if the destination is directly south of it, as shown by the source-destination pair (S_1, D_1), unless its west-going link is faulty (e.g., (S_3, D_3)). Overshooting adds tolerance for faulty link(s) directly south of a packet (e.g., (S_5, D_5) with a faulty output link of S_5). The same rule applies to the south forwarding direction as indicated by (S_4, D_4). After negative directions, the routing algorithm tries positive directions (i.e., east or north direction). No overshoot is needed for positive directions (e.g., (S_8, D_8)). The general rule is that a packet already traveling in the positive direction does not change to a negative direction unless there is no danger of forming a cyclic deadlock. For example, the last east-to-south turn of (S_2, D_2) is only allowed if it has no potential of forming a deadlock. If there were a packet in D_2, the packet would instead be dropped in order to prevent potentially creating a cycle of dependencies, which leads to a deadlock. This routing algorithm always routes the packet around a single fault, such as (S_2, D_2) and (S_6, D_6). The algorithm can often deliver a packet even in the presence of two link faults (e.g., (S_7, D_7)).

Table 1. Variables used in Algorithm 1 and Invariants 1–4.

Variable	Type	Definition
x, y	int	Current x and y coordinates of the packet
x', y'	int	Immediate next x and y coordinates of the packet
x_d, y_d	int	x and y coordinates of the packet's destination node
x_m, y_m	int	The maximal (corner) coordinates for a given NoC
dir	enum	Direction $dir \in \{n, e, s, w, i\}$, where i represents
		a newly injected packet with no traveling direction
f_{dir}	bool	Node (x, y) has a faulty link in the direction dir
τ	enum	The packet's current traveling direction, $\tau \in dir$
τ'	enum	The packet's next traveling direction, $\tau' \in dir$
$\nu(x, y)$	bool	Node (x, y) has a packet
$\mu(x, y)$	bool	Node (x, y) sent a packet one step previously
σ	list	Infinite repeated livelock pattern
$\mathcal{L}_\sigma$	set(list)	Set of all nodes constituting a livelock pattern σ

Algorithm 1: Link-Fault-Tolerant Routing Algorithm [29].

Input: x, y, τ
Output: x', y', τ'

1 **while** $\neg delivered$ **do**
2 **if** $x = x_d \wedge y = y_d$ **then**
3 $delivered := true$;
4 **else if** (x_d, y_d) *is 1 hop away and link is free* **then**
5 $x' := x_d; y' := y_d;$ ▷ Send to destination
6 **else if** $x \neq 0 \wedge \neg f_w \wedge (\tau \in \{w, s, i\}) \wedge ((x_d \leq x) \vee (y_d \geq y \wedge f_s))$ **then**
7 $x' := x - 1; y' := y, \tau' := w;$ ▷ Send west
8 **else if** $y \neq 0 \wedge \neg f_s \wedge (\tau \in \{s, w, i\}) \wedge ((y_d \leq y) \vee (x_d \geq x \wedge f_w))$ **then**
9 $x' := x; y' := y - 1, \tau' := s;$ ▷ Send south
10 **else if** $x \neq x_m \wedge \neg f_e \wedge \tau \neq w \wedge (x_d > x + 1 \vee (x_d > x \wedge y_d = y + 1))$ **then**
11 $x' := x + 1; y' := y, \tau' := e;$ ▷ Send east
12 **else if** $y \neq y_m \wedge \neg f_n \wedge \tau \neq s \wedge y_d > y$ **then**
13 $x' := x; y' := y + 1, \tau' := n;$ ▷ Send north
14 **else if** $x \neq 0 \wedge \neg f_w \wedge x_d \leq x \wedge (\tau \neq e \vee (y_d = y + 1 \wedge x_d = x))$ **then**
15 $x' := x - 1; y' := y, \tau' := w;$ ▷ Send west
16 **else if** $y \neq 0 \wedge \neg f_s \wedge y_d \leq y \wedge \tau \neq n$ **then**
17 $x' := x; y' := y - 1, \tau' := s;$ ▷ Send south
18 **else if**
 $x \neq x_m \wedge \neg f_e \wedge x_d \geq x \wedge (\tau \neq w \vee x_d = x \vee (x_d = x + 1 \wedge y_d \neq y + 1))$ **then**
19 $x' := x + 1; y' := y, \tau' := e;$ ▷ Send east
20 **else if** $y \neq y_m \wedge \neg f_n \wedge y_d \geq y \wedge (\tau \neq s \vee x_d \geq x)$ **then**
21 $x' := x; y' := y + 1, \tau' := n;$ ▷ Send north
22 **else**
23 *break*; ▷ Unroutable packet. Drop.

5 Refutation-Based Verification

We attempt to scale up verification of the correctness of Algorithm 1 by implementing it at the routing node level and stripping away architecture and communication details. Our aim is to check that Algorithm 1 is free of *livelock*. A major cause of livelock is excessive fault tolerance in the routing algorithm. Because every node attempts to provide an alternate route to a packet, it is possible that a packet could circle around several nodes without ever reaching its destination. As we later prove, livelock does not occur when a network contains less than two faults. While this algorithm was proven in [29] to be livelock-free on a 2×2 network, we discover additional livelock patterns which emerge upon scaling to a 3×3 NoC.

Part of the condition for livelock is a packet's inability to reach its destination. Livelock freedom in this network implies that a packet is either delivered or dropped (i.e. deemed unroutable) due to link-fault configurations. In both cases, a packet's trace is finite. This enables us to identify known non-livelock traces by simulating the routing algorithm for a finite number of moves. We use this observation to adapt a refutation-based simulation in which simulation

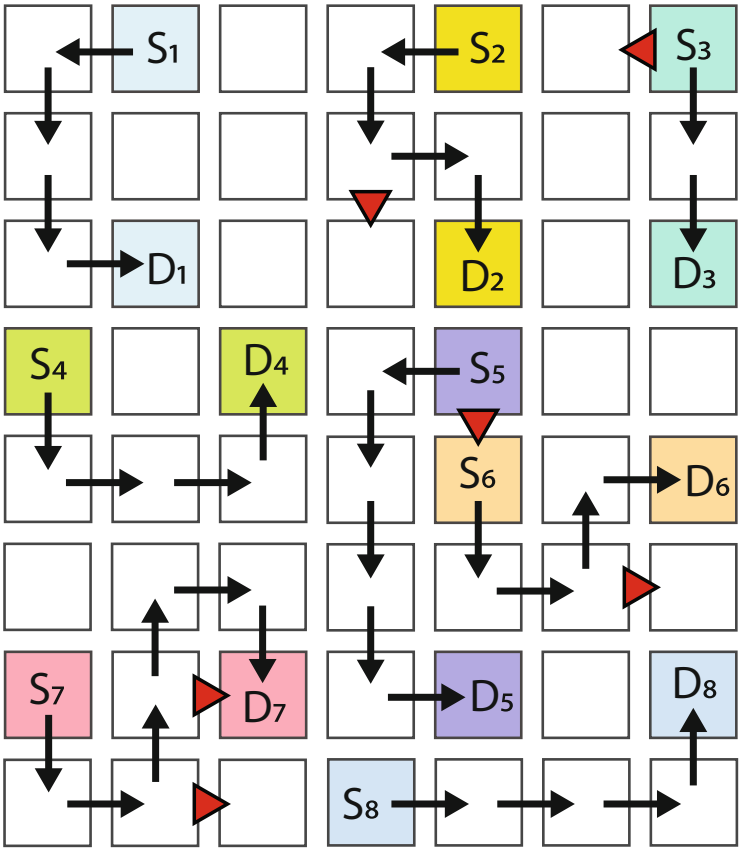

Fig. 1. Fault-tolerant routing examples. The black arrows show the path a packet will take; the red arrows point in the direction of a link fault. (Color figure online)

eliminates traces representing a packet being delivered or dropped. If a finite trace does not belong to either case, it is extracted to construct an invariant in IVy. This invariant is used to verify that the trace is truly livelock.

5.1 Disproving Livelock Through Refutation-Based Simulation

To quickly view the routes of each individual packet, we construct a C++ NoC model implementing Algorithm 1. Starting with a 3×3 NoC, we simulate every possible packet's route within K steps. K is an overestimate of the maximal number of steps that a packet requires to be delivered or dropped.

Our experiments are successful: a finite trace with $K = 1000$ can be efficiently generated when scaling up to a 15×15 NoC. A Windows 10 machine (version 1903) with a 2 GHz 4-Core CPU and 8 GB of memory simulates every possible packet's route and verifies livelock traces in under 9 h. This is a significant scale-up of livelock verification as compared to the 2×2 NoC verified in [29].

While $K = 1000$ is sufficient to identify potential livelock patterns on a massive network, this approach slows down greatly as the network's size increases. It is also based on finite dimensions provided at the time of simulation. Verification time increases exponentially upon scaling beyond a 10×10 network.

5.2 Proving Livelock Using IVy

For each trace that is not delivered or dropped within K steps, we developed a script to analyze the trace and identify looping behavior. Based on the packet's starting and destination coordinates, the configuration of faulty links, and the looping behavior, the script constructs an IVy model.

To effectively describe livelock, invariant checking in IVy begins within the livelock pattern. Since the simulation shows that the packet can enter the pattern, the only portion remaining to be checked is whether the livelock pattern is truly infinite. Using IVy's interactive proof assistant and the trace analysis script, the invariants described next are *automatically* constructed to check the infinite cycles of each trace.

Invariant 1 restricts the destination coordinates of the model to the destination coordinates of the trace produced by simulation. This eliminates every case where the destination node is not involved in producing livelock. Invariant 2 ensures that the only nodes which ever obtain the packet are nodes in $\mathcal{L}_\sigma$, the set of nodes in the livelock trace. We define the immediate precedence order $(x, y) \prec (x', y') \in \sigma$ as a predicate to formulate the following two invariants. These invariants together restrict invariant checking to each livelock trace. They define the order in which nodes receive the packet. Invariant 3 checks that every time a packet leaves a certain node, it will be traveling in the direction that leads to the next node in trace σ. Invariant 4 checks that every packet that leaves node (x, y) in σ is forwarded to its immediate next node (x', y') in σ. Enumerations of current and immediate next node coordinates for Invariants 3 and 4 are automatically added to the IVy model. Table 1 defines variables used in these invariants.

$$x_d = D_x \wedge y_d = D_y \tag{1}$$

$$\bigwedge_{0<x\leqslant x_m;\ 0<y\leqslant y_m} ((x, y) \notin \mathcal{L}_\sigma \wedge \neg\nu(x, y)) \tag{2}$$

$$\bigwedge_{0<x,x'\leqslant x_m;\ 0<y,y'\leqslant y_m} ((x, y), (x', y') \in \mathcal{L}_\sigma \wedge ((x, y) \prec (x', y') \in \sigma) \wedge (\nu(x', y') \implies \tau(x', y') = \tau'(x, y))) \tag{3}$$

$$\bigwedge_{0<x,x'\leqslant x_m;\ 0<y,y'\leqslant y_m} ((x, y), (x', y') \in \mathcal{L}_\sigma \wedge ((x, y) \prec (x', y') \in \sigma) \wedge (\mu(x, y) \implies \nu(x', y') \wedge \tau(x', y') = \tau'(x, y))) \tag{4}$$

These invariants reproduce in IVy the potential livelock pattern obtained from simulating the C++ model. If IVy confirms that they are inductive invariants for the corresponding model, it proves not only that the trace σ is possible but that once it begins, no single move can remove the packet from σ. Therefore, σ is infinite, indicating a true livelock pattern.

This refutation-based verification method combines the efficiency of C++ routing simulation with the power of IVy invariant checking. It streamlines trace extraction and pruning from simulation and invariant formulation for proving the existence of livelock. We have automated this process, and it has demonstrated substantial improvement in scaling livelock checking to significantly larger networks than the 2×2 NoC proven in [29], in our case, up to 15×15.

6 User-Aided Livelock Removal

After confirming livelock traces using IVy's invariant verification, we correct the routing logic in Algorithm 1 to prevent each livelock pattern as *early* in a packet's journey as possible. A script automatically identifies each livelock pattern to aid the user in adjusting the algorithm. Our incremental livelock removal process starts with a 3×3 NoC and scales up once livelock freedom is achieved. We first analyze the decision that initiates livelock as illustrated by the blue arrows in Figs. 2 and 3.

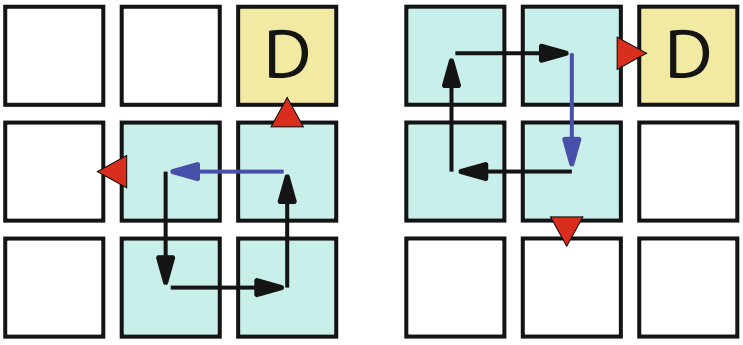

Fig. 2. Livelock patterns on a 3×3 NoC.

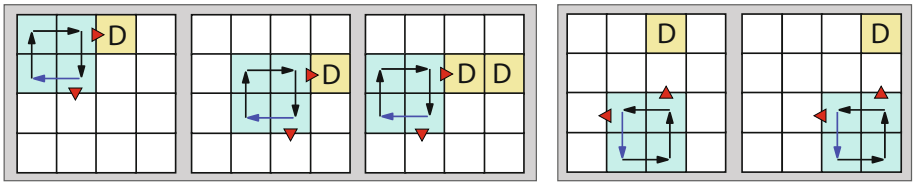

Fig. 3. Patterns from the 4×4 NoC.

Our observation is that the majority of livelock patterns that emerge in a 3×3 NoC are almost identical to those in Fig. 2. To remove these patterns, we modify the algorithm iteratively at each condition. The modification that removes the largest quantity of livelock traces while maintaining a low number of unroutable packets is then tested as the current working model. This process repeats until the NoC is livelock-free.

By analyzing the identified livelock patterns in Fig. 2, we find that the livelock scenario on the left can be removed by adding the condition ($x_d \neq x + 1 \vee y_d \neq y + 1$) to line 8 of Algorithm 1, effectively preventing the packet from being routed south if the destination is one node northeast. We continue removing other livelock patterns using the steps below.

1. Prune the prefix of a livelock trace and then identify each turn a packet makes in a livelock cycle.
2. Manually adjust the conditions for one turn at a time by excluding the current state of the packet at a specific node from making the livelock-producing turn. These conditions include the packet's current location and traveling direction, the arrangement of link faults, and the destination coordinates.
3. Simulate the model with each modification to the routing algorithm.
4. Identify and count the potential livelock scenarios that were removed. Since modifications decrease a packet's ability to move, no new livelock scenarios are created and the process can terminate.
5. Implement the modification that yields the *fewest* livelock scenarios and unroutable cases to the routing algorithm, effectively removing the livelock pattern under analysis as well as other similar patterns.
6. Simulate the modified model and repeat all steps above to eliminate any remaining livelock scenarios.

A user can intuitively find similarities between livelock patterns. This interactive procedure vastly simplifies routing algorithm modification, especially when scaling up the network. For example, when we discover that the livelock patterns on a 4×4 network very closely resemble those of the 3×3 network, it is clear that the modification to the algorithm has to account for scenarios in different areas of the NoC. Figure 3 shows two groups of livelock patterns that emerge when scaling up to a 4×4 NoC. The nodes marked D represent possible locations for a destination that caused livelock. After analyzing each move of the new livelock patterns, the previously mentioned modification to the routing protocol is designed to be scalable, and packets are excluded from travelling south if the destination is one node east and any number of nodes north of the packet. This incremental process repeats to produce a livelock-free routing algorithm.

Specifically, conditions 1, 2, and 3 shown below have been added as *conjunctive* clauses to lines 6, 8, and 16 of Algorithm 1, respectively. The addition of these conditions produces a livelock-free routing algorithm for up to a 15×15 NoC. Because a packet cannot be in livelock if it is delivered or dropped within K steps, and because *all* traces produced by the C++ model show that a packet is delivered or dropped within K steps, no livelock exists in the 15×15 NoC.

1. $y_d \neq y + 1 \vee \tau \neq s$
2. $x_d \neq x + 1 \vee y_d < y + 1$
3. $\tau \neq e \vee x_d \neq x_m \vee y_d \neq y_m$

7 Unroutable Packets

Modifying the original routing algorithm to remove livelock can turn an infinite livelock pattern into a finite trace representing either packet delivery or unroutability. The "unroutable" status of a packet is determined when a routing node has exhausted all alternative routes for a packet but still cannot deliver it. Labeling a packet as unroutable allows it to be removed from the network, which reduces unnecessary network traffic that consumes power as is the case for livelock. On the other hand, minimizing traces for unroutable packets while removing livelock scenarios is ideal as it allows improved packet delivery rate for a NoC while reducing unprofitable network traffic. The C++ model tracks unroutable packets in addition to potential livelock patterns.

We find that some unroutable cases are introduced when livelock is removed. However, this modification also allows for many packets otherwise in livelock to be converted into deliverable packets. An interesting cause of packet unroutability is livelock prevention. This can cause seemingly random patterns of unroutable packets, but they form an insignificant amount of packets, especially as the network is scaled. In these cases, the faulty links are in relatively unique arrangements in the network.

8 Zone-Based Routing Model

The aforementioned refutation-based livelock verification and removal method can be used to identify and eliminate livelock and produce an improved routing algorithm. It, however, becomes unrealistic to simulate when livelock checking scales to a large NoC. A key observation during the livelock removal process is that traces representing deliverable, unroutable, and livelock scenarios share common features and can be abstracted for more efficient analysis. A critical observation of the routing algorithm is formed after grouping and classifying these traces: the *relative* positions between a packet's source and destination nodes can represent their corresponding absolute coordinate-based positions. Therefore, the coordinate-based NoC that has been discussed so far can be lifted to a NoC of arbitrary size as the relative source-destination positions no longer require the coordinates. This leads to further simplification of the model: a packet used to store current and destination coordinates now only stores its zone information.

Predicates and Invariants for Routing Zones. It is imperative that the abstraction of a model accurately represent the model. We utilize IVy to aid in incrementally creating the zone-based routing abstraction. Each location-based

condition from the routing algorithm is extracted and mapped to a predicate variable. If two physically adjacent zones satisfy an identical set of predicates, the zones are merged. The detailed process is as follows.

First, we extract location-based conditions from the routing algorithm to form predicates, e.g., the condition $y_d \geqslant y$ in line 6 of Algorithm 1 is represented by a predicate variable p_1. A zone represents a set of routing nodes satisfying the same predicates. Denote zone $\vec{Z}_i = \langle p_1, \ldots, p_n \rangle$ as a vector of n predicate variables, i.e., location-based predicates extracted from the routing algorithm. In the case of Algorithm 1, $n = 18$. Even if one predicate can cover the other, predicates are stored as separate variables. For example, if p_1 is $x_d = x$ and p_2 is $x_d \geqslant x$, we denote p_1 and p_2 as separate predicates. Given a NoC routing algorithm and a set of predicates, we formulate the three invariants listed below and check them against our NoC model in IVy:

1. $\forall i \neq j : \vec{Z}_i \neq \vec{Z}_j$ (Every zone is unique.)
2. $\forall i, \exists k, s.t. \vec{Z}_i[k] = true$ (Every zone has at least one true predicate.)
3. $\forall k, \exists i, s.t. \vec{Z}_i[k] = true$ (Every predicate is true in at least one zone.)

When Invariant 1 fails, there must exist one pair of zones, namely, $\vec{Z}_i$ and $\vec{Z}_j$, that are identical (i.e., $\vec{Z}_i = \vec{Z}_j$). When Invariant 1 holds, every zone is unique and no two zones can be merged. Invariants 2 and 3 describe the necessity of each zone and each predicate, respectively. That is, if a predicate evaluates to false everywhere in the NoC, it can be removed from the routing algorithm.

Abstracting Routing Nodes into Zones. We begin with a 1×1 NoC with one zone, i.e., the destination zone shown in Fig. 4(a). Note that we represent the destination node with white text on a black background in all subfigures of Fig. 4. Because every zone in a 1×1 NoC is trivially unique, we scale up by adding one node in each direction to create a 3×3 NoC. When encountering a new node after expanding the NoC for the first time, we assign each node a unique zone identifier, as shown in Fig. 4(b). Then we check the three invariants above against the IVy model. When Invariant 1 fails, IVy returns two identical zones as a counterexample. When Invariant 2 or 3 fail, IVy returns the unused zone or predicate to be evaluated by the user. If equivalent zones are physically adjacent on the NoC, we merge them into one zone then repeat checking these invariants. When Invariant 1 holds, it indicates that no further abstraction can be made as every zone is unique, and combining two unique zones would cause neighboring nodes within the same zone to forward a packet differently. For the 3×3 NoC, IVy proves that every zone is unique. Therefore, we continue scale it up to a 5×5 network as shown in Fig. 4(c). Invariant checking in IVy shows that some, but not all of the new zones are unique. For instance, after IVy finds that $Z_H = Z_X$, they are merged into one zone Z_L. This process is repeated until all zones are unique. The resulting zones are shown in Fig. 4(d).

Before further scaling the zone-based model, we first analyze the routing decision conditions of Algorithm 1. We observe that the routing decision conditions are reliant only on the following distance specifications (where m represents

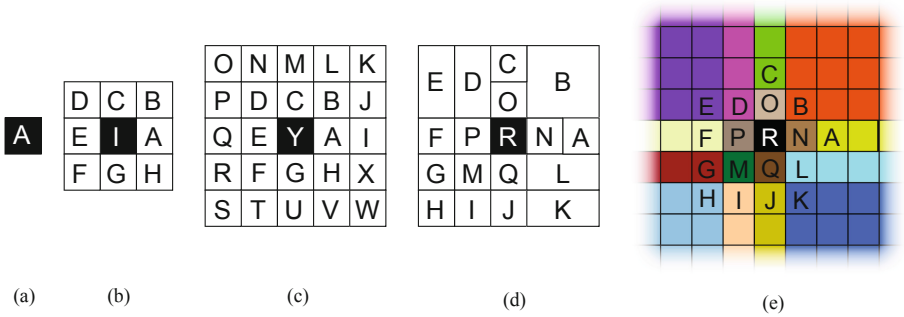

Fig. 4. Development of abstract zones

either an x-coordinate or a y-coordinate): $m < m_d - 1$, $m < m_d$, $m \leqslant m_d$, $m = m_d$, $m \geqslant m_d$, $m > m_d$, and $m > m_d + 1$. Because none of these specifications differentiates between a node that is two or more nodes away from the destination, the nodes or zones on an edge in a 5×5 network can be extended to represent all nodes more than one node away from the destination. As an example, the node labeled C in Fig. 4(d) is representative not only for the node two nodes north of the destination, but also for *all* nodes more than one node north. To validate our conjecture, we use IVy to check the aforementioned invariants in arbitrarily large networks where all new zones are present. IVy confirms all zones are unique. Figure 4(e) and Table 2 show the final zones for Algorithm 1.

Table 2. Formulas corresponding to each zone in Fig. 4(e)

Zone	x	y	Size	Zone	x	y	Size
A	$x > x_d + 1$	$y = y_d$	$k \times 1$	J	$x = x_d$	$y < y_d - 1$	$1 \times k$
B	$x > x_d$	$y > y_d$	$k \times k$	K	$x > x_d$	$y < y_d - 1$	$k \times k$
C	$x = x_d$	$y > y_d + 1$	$1 \times k$	L	$x > x_d$	$y = y_d - 1$	$k \times 1$
D	$x = x_d - 1$	$y > y_d$	$1 \times k$	M	$x = x_d - 1$	$y = y_d - 1$	1×1
E	$x < x_d - 1$	$y > y_d$	$k \times k$	N	$x = x_d + 1$	$y = y_d$	1×1
F	$x < x_d - 1$	$y = y_d$	$k \times 1$	O	$x = x_d$	$y = y_d + 1$	1×1
G	$x < x_d - 1$	$y = y_d - 1$	$k \times 1$	P	$x = x_d - 1$	$y = y_d$	1×1
H	$x < x_d - 1$	$y < y_d - 1$	$k \times k$	Q	$x = x_d$	$y = y_d - 1$	1×1
I	$x = x_d - 1$	$y < y_d - 1$	$1 \times k$	R	$x = x_d$	$y = y_d$	1×1

9 Zone-Based IVy Implementation

In the zone-based model, it is no longer necessary to maintain the coordinates for routing nodes. In order for the routing algorithm to determine a packet's next forwarding direction, one needs to know the following information about a

packet: current traveling zone ⊞, current traveling direction τ, and the status of the current routing node's four output links Λ_{dir}, where $dir \in \{w, s, n, e\}$, which can be free (✓), faulty (⊠), or edge (⊥). The "edge" status of an output link indicates that the current traveling node is on an edge of a NoC. Table 3 lists these variables and their types. Note that τ is the most recent direction of travel or the direction of travel that led into the current node. Since coordinates are not used in the zone-based model, τ is no longer associated to its node's coordinates, but rather an enumerative typed variable.

Table 3. Variables for zone-based model.

Variable	Type	Definition
⊞	enum	Packet's most recent traveling zone (A to R) in Fig. 4(e)
τ	enum	Packet's most recent traveling direction, $\tau \in dir$
Λ_{dir}	enum	Output link in the direction specified by dir, where $dir = \{n, e, s, w\}$; and output link can be free (✓), faulty (⊠), or edge (⊥)

On-the-fly NoC Construction. The zone-based model inherits the assumption that the number of faults in a network does not exceed two. This number could be modified to improve the protocol and guarantee a higher fault tolerance, but increasing fault tolerance creates additional complexity without necessarily increasing packet's ability to route [29]. Thus, the zone-based model aims to guarantee livelock freedom for two-fault tolerant routing. We specify the zone-based routing algorithm in IVy. In an arbitrarily large NoC, the user is no longer required to specify dimensions for the network. Instead, a network is constructed on-the-fly while the packet travels. Forwarding decisions in the abstract network use the following procedure:

1. The current zone of a packet (⊞) is nondeterministically chosen.
2. The current node's link statuses ($\Lambda_{n,e,s,w}$) are nondeterministically chosen.
3. The routing algorithm determines the direction to forward the packet.
4. The packet is forwarded and the process repeats.

To guarantee realistic scenarios throughout the IVy NoC model, we specified constraints and heuristics to aid the nondeterministic choices in the procedure above. For example, if a packet is sent west from an unknown location in zone K (see Fig. 4), a westbound move can keep it in zone K or forward it to zone J, decided nondeterministically. However, if a packet travels from zone N to zone B in one move and then travels west one node, it must be in zone O. The routing model also excludes impossible edge cases. For instance, the rules 5 and 6 below are assumptions for the east links, and symmetric rules apply to the other links. As shown in Fig. 6, this requires that the edges of the network remain intact. It removes unrealistic scenarios including one in which a packet reaches the east edge, travels south, and then is allowed to be forwarded east again. It disallows the scenario on the left of Fig. 6 but enforces the scenario on the right. While they do not define perfect conditions for east links, they allow every possible scenario to be tested, along with some impossible scenarios. Since

realistic scenarios are a subset of all verified scenarios, and the set of all verified scenarios is livelock-free, we prove livelock freedom in all realistic scenarios.

$$(\tau \in \{n, s\} \wedge \Lambda_e = \perp) \implies (\Lambda'_e = \perp) \tag{5}$$

$$((\tau \in \{n, s\} \wedge \Lambda_e \neq \perp) \vee \tau = w) \implies (\Lambda'_e \neq \perp) \tag{6}$$

Abstract Moves. In order to effectively check for a livelock pattern consisting of a large number of single-step moves of a packet, we specify an *abstract move*, which aggregates a sequence of consecutive stuttering single-step moves into a single move. It is the abstract moves of a packet that are stored as it travels. For example, if a packet takes 300 moves east in zone E before turning south, but remaining in E, they are all represented by one abstract east move followed by another abstract move to the south. Define the state of the zone-based model as $(\boxplus, \tau, \Lambda_w, \Lambda_s, \Lambda_n, \Lambda_e)$. A single-step move α causes possibly trivial update to the current state, i.e., $(\boxplus, \tau, \Lambda_w, \Lambda_s, \Lambda_n, \Lambda_e) \xrightarrow{\alpha} (\boxplus', \tau', \Lambda'_w, \Lambda'_s, \Lambda'_n, \Lambda'_e)$. A single-step move is *stuttering*, denoted as ε, if $\tau' = \tau$, i.e., the move does not change the packet traveling direction. An abstract move is used to represent a (finite) sequence of stuttering single-step moves and a sequence of abstract moves only consists of non-stuttering moves. As a packet travels, an abstract move is formed by storing only the end state of a sequence of stuttering moves. Specifically, given a sequence of moves, $s_i \xrightarrow{\alpha} s_{i+1} \xrightarrow{\varepsilon} s_{i+2} \xrightarrow{\varepsilon} \ldots, \xrightarrow{\varepsilon} s_{i+k} \xrightarrow{\alpha} s_{i+(k+1)}$, it is abstracted as $s_i \xrightarrow{\alpha} s_{i+k} \xrightarrow{\alpha} s_{i+(k+1)}$. For instance, for a packet with the following sequence of exact moves, $(B, w, \checkmark, \boxtimes, \checkmark, \checkmark) \xrightarrow{\varepsilon} (B, w, \checkmark, \checkmark, \checkmark, \checkmark) \xrightarrow{\alpha} (B, s, \checkmark, \checkmark, \checkmark, \checkmark)$, its abstract sequence becomes $(B, w, \checkmark, \checkmark, \checkmark, \checkmark) \xrightarrow{\alpha} (B, s, \checkmark, \checkmark, \checkmark, \checkmark)$. Storing only abstract moves while checking for livelock enables us to detect very large cyclic patterns. As shown in Fig. 5, we can detect cycles using only the turns in those cycles. Using abstract moves significantly reduces verification effort while preserving livelock in the system. Since livelock is based on a series of moves that change traveling directions, the number of stuttering moves a packet makes in sequence does not impact livelock.

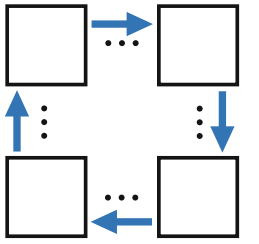

Fig. 5. Abstract moves

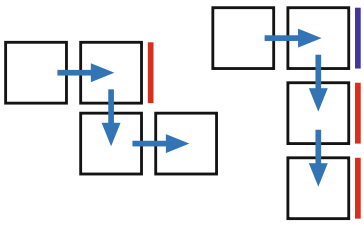

Fig. 6. Edge heuristics

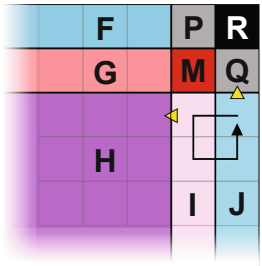

Fig. 7. Livelock pattern

10 Verification of Livelock Freedom

The invariants used for livelock checking are presented in Fig. 8. Let τ_n represent the traveling direction of the n^{th} stored abstract move along a packet trace to destination, with $n = 0$ indicating the most recent decision and $n = 6$ indicating the earliest recorded abstract move. The upper bound of stored abstract moves T_{max} is set to 7 because our experimentation indicates that it is the lowest number that does not cause "false positives" – packets traveling in a near-complete loop without entering livelock. Since a livelock pattern is infinite but must contain at least 2 nodes (in a back-and-forth livelock pattern) or 4 nodes (in a cyclical pattern), T_{max} has to be at least 4. When $4 \leqslant T_{max} \leqslant 6$, scenarios emerge that cause a packet to complete a loop without entering livelock. For example, a packet may travel to the destination by passing through zones $J \rightarrow Q \rightarrow M \rightarrow I \rightarrow J \rightarrow K \rightarrow N \rightarrow R$ when the node at zone M is on the west edge and has a faulty north link. Note that zone L is not stored between $K \rightarrow N$ because the packet traveling direction remains the same from K through L until it turns west in zone N. While not a livelock pattern, it includes four nodes that checking with $T_{max} = 4$ would cause to be flagged as a livelock pattern. Clockwise, counter-clockwise, and back-and-forth livelock patterns are given as invariants 7 to 10, 11 to 14, and 15 to 18 in Fig. 8, respectively. These twelve invariants cover all possible livelock patterns in the zone-based model and therefore, are used for livelock detection and removal in our work.

For livelock freedom verification in the zone-based model, our approach offers a *stronger* guarantee than livelock freedom. That is, in addition to proving the routing algorithm is free of livelock, we prove that it is free of any traces making more than seven abstract turns in a cyclic pattern. If a packet does not reach its destination without making seven cyclic abstract turns, then its abstract trace is considered as a "livelock" trace, even though it may have a chance of reaching its destination after seven abstract turns. Such a trace is automatically detected and then used for improving the routing algorithm as discussed later in this section. Thus, we argue that the network is efficient since packets mostly take a direct path to the destination and avoid traveling needlessly in cyclical patterns.

Verification of the zone-based routing algorithm is performed in IVy with the ABC implementation of Property-Directed Reachability [2]. After detecting the first invariant violation, the model checker terminates and returns a counterexample representing a livelock trace. For the purpose of correcting Algorithm 1 to achieve livelock-free routing, it is required to collect all livelock traces. To enumerate every possible livelock scenario, we automate incremental livelock trace generation by iteratively adding a previously generated livelock trace as a new IVy invariant and then invoking IVy to find the next livelock pattern. A report of each livelock pattern is generated to aid a user in adapting the routing algorithm to remove it. The report makes clear which routing decision is taken. The traces provided by this model are sufficiently informative for a user to correct the routing algorithm.

We begin livelock verification by first encoding in IVy all invariants shown in Fig. 8. Each invariant in this figure represents a zone-independent packet trav-

$$\tau_6 = s \wedge \tau_5 = w \wedge \tau_4 = n \wedge \tau_3 = e \wedge \tau_2 = s \wedge \tau_1 = w \wedge \tau_0 = n \quad (7)$$
$$\tau_6 = w \wedge \tau_5 = n \wedge \tau_4 = e \wedge \tau_3 = s \wedge \tau_2 = w \wedge \tau_1 = n \wedge \tau_0 = e \quad (8)$$
$$\tau_6 = n \wedge \tau_5 = e \wedge \tau_4 = s \wedge \tau_3 = w \wedge \tau_2 = n \wedge \tau_1 = e \wedge \tau_0 = s \quad (9)$$
$$\tau_6 = e \wedge \tau_5 = s \wedge \tau_4 = w \wedge \tau_3 = n \wedge \tau_2 = e \wedge \tau_1 = s \wedge \tau_0 = w \quad (10)$$
$$\tau_6 = e \wedge \tau_5 = n \wedge \tau_4 = w \wedge \tau_3 = s \wedge \tau_2 = e \wedge \tau_1 = n \wedge \tau_0 = w \quad (11)$$
$$\tau_6 = n \wedge \tau_5 = w \wedge \tau_4 = s \wedge \tau_3 = e \wedge \tau_2 = n \wedge \tau_1 = w \wedge \tau_0 = s \quad (12)$$
$$\tau_6 = w \wedge \tau_5 = s \wedge \tau_4 = e \wedge \tau_3 = n \wedge \tau_2 = w \wedge \tau_1 = s \wedge \tau_0 = e \quad (13)$$
$$\tau_6 = s \wedge \tau_5 = e \wedge \tau_4 = n \wedge \tau_3 = w \wedge \tau_2 = s \wedge \tau_1 = e \wedge \tau_0 = n \quad (14)$$
$$\tau_6 = w \wedge \tau_5 = e \wedge \tau_4 = w \wedge \tau_3 = e \wedge \tau_2 = w \wedge \tau_1 = e \wedge \tau_0 = w \quad (15)$$
$$\tau_6 = e \wedge \tau_5 = w \wedge \tau_4 = e \wedge \tau_3 = w \wedge \tau_2 = e \wedge \tau_1 = w \wedge \tau_0 = e \quad (16)$$
$$\tau_6 = n \wedge \tau_5 = s \wedge \tau_4 = n \wedge \tau_3 = s \wedge \tau_2 = n \wedge \tau_1 = s \wedge \tau_0 = n \quad (17)$$
$$\tau_6 = s \wedge \tau_5 = n \wedge \tau_4 = s \wedge \tau_3 = n \wedge \tau_2 = s \wedge \tau_1 = n \wedge \tau_0 = s \quad (18)$$

Fig. 8. Livelock patterns encoded as invariants for routing zones.

eling pattern. Therefore, such a pattern can exist in a number of livelock traces when mapped to actual regions. Our technique relies on IVy to incrementally enumerate actual livelock traces. For example, Invariant 14 in Fig. 8 is encoded as $\neg\sigma_0$ below. Then IVy can detect a violation of $\neg\sigma_0$ by returning an actual livelock trace as a counterexample shown as σ_1 below. This trace describes packet travel between zones I and J, where `sentN` represents τ_N and Λ_{dirN} is represented by `northLinkN`, `eastLinkN`, and so forth. A packet's X^{th} zone $\boxplus_X$ is given by `packet.znX` and `packet.zn3 = i` means that the packet's third zone was I. More concrete examples are found on GitHub (See footnote 1).

$\neg\sigma_0$:
```
~(packet.sent6 = south & packet.sent5 = east & packet.sent4
= north & packet.sent3 = west & packet.sent2 = south &
packet.sent1 = east & packet.sent0 = north)
```

σ_1 :
```
(packet.zn6 = i & packet.zn5 = j & packet.zn4 = j &
packet.zn3 = i & packet.zn2 = i & packet.zn1 = j
& node.southLink5 = edge & node.eastLink4 = edge &
node.southLink4 = edge & node.northLink3 = faulty &
node.eastLink3 = edge & node.westLink2 = faulty &
node.southLink1 = edge & node.eastLink = edge &
node.southLink = edge)
```

The amended invariant is constructed as $\neg\sigma_0 \vee \sigma_1$ (i.e., $\sigma_0 \Rightarrow \sigma_1$) so that IVy can skip livelock trace σ_1 and continue to search for the next trace. When n livelock traces are produced, the amended invariant is constructed as $\neg\sigma_0 \vee \sigma_1 \vee \sigma_2 \vee \cdots \vee \sigma_{n-1} \vee \sigma_n$. This process is repeated for verifying the model and

appending new invariants representing livelock traces as disjunctive clauses to existing ones until every livelock pattern has been identified, when verification terminates with a livelock-free model.

10.1 Incremental Removal of Livelock Traces in Routing Algorithm

We apply a similar method to that presented in Sect. 6 to improve the routing algorithm until no livelock patterns can be found. For instance, consider the livelock pattern from Fig. 7. To eliminate this pattern, we remove the condition $\tau \neq e \vee x_d \neq x_m \vee y_d \neq y_m$ (implemented in Sect. 6) from the routing algorithm. We then add to the first west decision a condition that truly eliminates livelock scenarios: $\boxplus \notin \{J, Q\} \vee \tau \neq n$, as shown on line 20 of Algorithm 2. In most cases, this does not produce additional unroutable scenarios, as a packet that satisfies the new condition (excluding it from the second west routing option) generally satisfies the condition to be routed east and then north or south toward the destination. This approach leads to the creation of our final zone-based livelock-free routing algorithm shown in Algorithm 2.

10.2 Verification Results for Zone-Based Routing Algorithm

The final zone-based link-fault-tolerant routing algorithm was verified to satisfy the invariants in Fig. 8 on a Windows 10 machine (version 1903) with an Intel Core i7 4-Core 2 GHz Processor and 8 GB memory. With a *single* faulty link, the improved routing algorithm is proven to be livelock-free with no unroutable packets. Under two-faulty-link configurations, this routing algorithm is also livelock-free, with only a small number of unroutable patterns. Because it is unlikely to find more than two faults on a NoC [29], only networks with one and two faults are tested. With no livelock detected for Algorithm 2, it runs in approximately four minutes. The original routing algorithm contains 18 livelock patterns which are discovered in under three hours. The livelock verification script can consistently detect livelock at a rate of under ten minutes per livelock scenario. Thus, the zone-based IVy verification is both more efficient and more accurate than the C++ simulation.

10.3 Detecting Unroutable Packets

Ideally, a fault-tolerant algorithm should be free of livelock while producing the fewest unroutable cases. Thus, it is important to identify unroutable packets on Algorithm 2. The same script that modifies the livelock-resistance invariants can be used to detect unroutable packets.

While it is difficult to obtain a finite percentage of unroutable packets on an arbitrarily large network, we can detect and count the patterns that cause a packet to become unroutable. In a similar fashion to livelock detection, the invariant `¬packet.unroutable` can be checked each time we verify the algorithm. When IVy finds a counterexample (i.e., a trace showing an unroutable packet) the script analyzes it and adds the trace of the scenario that caused an unroutable

Algorithm 2: Final Livelock-Free Zone-Based Routing Algorithm.

Input: $\boxplus, \Lambda_{dir}, \tau$
Output: τ'

while $\neg delivered$ **do**
  if $\boxplus = R$ **then**
    $delivered := true$;
  else if $\boxplus = N \wedge \Lambda_w = \checkmark$ **then**
    $\tau' := w$; ▷ Send west
  else if $\boxplus = O \wedge \Lambda_s = \checkmark$ **then**
    $\tau' := s$; ▷ Send south
  else if $\boxplus = P \wedge \Lambda_e = \checkmark$ **then**
    $\tau' := e$; ▷ Send east
  else if $\boxplus = Q \wedge \Lambda_n = \checkmark$ **then**
    $\tau' := n$; ▷ Send north
  else if $\Lambda_w = \checkmark \wedge \tau \in \{w, s, i\} \wedge (\boxplus \in \{A, B, C, J, K, L, N, O, Q\} \vee (\boxplus \notin \{D, E\} \wedge \Lambda_s = \boxtimes)) \wedge (\boxplus \notin \{G, L, M, Q\} \vee \tau \neq s)$ **then**
    $\tau' := w$; ▷ Send west
  else if $\Lambda_s = \checkmark \wedge \tau \in \{w, s, i\} \wedge (\boxplus \in \{A, B, C, E, F, N, O, P\} \vee (\boxplus \notin \{D, K, L\} \wedge \Lambda_w = \boxtimes))$ **then**
    $\tau' := s$; ▷ Send south
  else if $\Lambda_e = \checkmark \wedge \tau \neq w \wedge \boxplus \in \{E, F, G, H, M\}$ **then**
    $\tau' := e$; ▷ Send east
  else if $\Lambda_n = \checkmark \wedge \tau \neq s \wedge \boxplus \in \{G, H, I, J, K, L, M, Q\}$ **then**
    $\tau' := n$; ▷ Send north
  else if $\Lambda_w = \checkmark \wedge \boxplus \in \{A, B, C, J, K, L, N, O, Q\} \wedge (\boxplus = Q \vee \tau \neq e) \wedge (\boxplus \notin \{J, Q\} \vee \tau \neq n)$ **then**
    $\tau' := w$; ▷ Send west
  else if $\Lambda_s = \checkmark \wedge \tau \neq n \wedge \boxplus \in \{A, B, C, D, E, F, N, O, P\}$ **then**
    $\tau' := s$; ▷ Send south
  else if $\Lambda_e = \checkmark \wedge \boxplus \in \{C, D, E, F, G, H, I, J, M, O, P, Q\} \wedge (\tau \neq w \vee \boxplus \in \{C, J, O, Q, D, I, P\})$ **then**
    $\tau' := e$; ▷ Send east
  else if $\Lambda_n = \checkmark \wedge \boxplus \in \{A, F, G, H, I, J, K, L, M, N, P, Q\} \wedge (\tau \neq s \vee \boxplus \in \{F, G, H, I, J, M, P, Q\})$ **then**
    $\tau' := n$; ▷ Send north
  else
    break; ▷ Unroutable packet. Drop

packet as a disjunctive clause to the original invariant. IVy can detect all of the unroutable scenarios within several hours using the same machine described in Sect. 10.2. The script generates a log file with traces of those unroutable patterns.

11 Conclusion

This paper describes a process for scalable verification of a fault-tolerant routing algorithm. While refutation-based simulation enables the model to scale from

2×2 to 15×15 NoCs, this approach allows us to discover livelock traces missed by the previous verification approach from [29]. We then propose an abstract zone-based routing algorithm model based on a packet's relative position to its destination. It uses Property-Directed Reachability to verify livelock freedom on arbitrarily large NoCs. We propose iterative techniques to automatically discover all livelock patterns. We use these patterns to derive an improved link-fault-tolerant routing algorithm that is livelock-free for arbitrarily large NoCs. This livelock freedom guarantees increase dependability of the analyzed fault-tolerant algorithm for its application in safety-critical systems.

Techniques developed in this paper are applicable to formal specification and verification of a variety of fault-tolerant adaptive routing algorithms, as well as, other NoC topologies. Future work includes investigating techniques for optimizing the zone-based, livelock-free routing algorithm to produce the minimal number of unroutable packets. We also plan to investigate other safety properties on the zone-based model, such as deadlock freedom.

Acknowledgment. The authors would like to thank Ken McMillan for his assistance in understanding the IVy formal specification language and utilizing the IVy verification tool. Landon Taylor was supported in part by National Science Foundation grant (CAREER-1253024). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors would also like to thank Adobe Systems Incorporated for their support.

References

1. Alhussien, A., Verbeek, F., van Gastel, B., Bagherzadeh, N., Schmaltz, J.: Fully reliable dynamic routing logic for a fault-tolerant NoC architecture. J. Integr. Circuits Syst. **8**(1), 43–53 (2013)
2. Berkeley Logic Synthesis and Verification Group: ABC: A system for sequential synthesis and verification (September 2020). http://www.eecs.berkeley.edu/~alanmi/abc/ and https://github.com/berkeley-abc/abc
3. Borrione, D., Helmy, A., Pierre, L., Schmaltz, J.: A formal approach to the verification of networks on chip. EURASIP J. Embed. Syst. **2009**(1), 1–14 (2009). https://dl.acm.org/doi/10.1155/2009/548324
4. Boutekkouk, F.: Formal specification and verification of communication in Network-on-Chip: an overview. Int. J. Recent Contrib. Eng. Sci. IT (iJES) **6**(4), 15–31 (2018). https://doi.org/10.3991/ijes.v6i4.9416
5. Bozga, M., Graf, S., Mounier, L.: IF-2.0: a validation environment for component-based real-time systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 343–348. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_26
6. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
7. Champelovier, D., et al.: Reference manual of the LNT to LOTOS translator (version 6.0). INRIA/VASY/CONVECS (June 2014)

8. Chatterjee, S., Kishinevsky, M., Ogras, U.Y.: Quick formal modeling of communication fabrics to enable verification. In: 2010 IEEE International High Level Design Validation and Test Workshop (HLDVT), pp. 42–49 (2010). https://doi.org/10.1109/HLDVT.2010.5496662
9. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_14
10. Dridi, M., Lallali, M., Rubini, S., Singhoff, F., Diguet, J.P.: Modeling and validation of a mixed-criticality NoC router using the IF language. In: Proceedings of the 10th International Workshop on Network on Chip Architectures. NoCArc 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3139540.3139543
11. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013). https://doi.org/10.1007/s10009-012-0244-z
12. van Gastel, B., Schmaltz, J.: A formalisation of xMAS. In: Gamboa, R., Davis, J. (eds.) Proceedings International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2013, Laramie, Wyoming, USA, 30–31 May 2013, EPTCS, vol. 114, pp. 111–126 (2013). https://doi.org/10.4204/eptcs.114.9
13. Helmy, A., Pierre, L., Jantsch, A.: Theorem proving techniques for the formal verification of NoC communications with non-minimal adaptive routing. In: DDECS, pp. 221–224. IEEE (2010)
14. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
15. Imai, M., Yoneda, T.: Improving dependability and performance of fully asynchronous on-chip networks. In: Proceedings of the 2011 17th IEEE International Symposium on Asynchronous Circuits and Systems, pp. 65–76. ASYNC 2011, IEEE Computer Society, Washington, DC, USA (2011). https://doi.org/10.1109/ASYNC.2011.15, http://dx.doi.org/10.1109/ASYNC.2011.15
16. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
17. Lecler, J.J., Baillieu, G.: Application driven network-on-chip architecture exploration & refinement for a complex SoC. Des. Autom. Embed. Syst. **15**, 133–158 (2011). https://doi.org/10.1007/s10617-011-9075-5, https://link.springer.com/article/10.1007/s10617-011-9075-5
18. McMillan, K.L.: IVy (2019). http://microsoft.github.io/ivy/, https://github.com/kenmcmil/ivy
19. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 190–202. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_12
20. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
21. Tsai, W.C., Lan, Y.C., Hu, Y.H., Chen, S.J.: Networks on chips: structure and design methodologies. J. Electr. Comput. Eng. **2012**, 15 (2012). https://doi.org/10.1155/2012/509465

22. Verbeek, F., Schmaltz, J.: A decision procedure for deadlock-free routing in wormhole networks. IEEE Trans. Parallel Distrib. Syst. **25**(8), 1935–1944 (2014). https://doi.org/10.1109/TPDS.2013.121
23. Verbeek, F., Schmaltz, J.: Automatic verification for deadlock in Networks-on-Chips with adaptive routing and wormhole switching. In: Proceedings of the fifth ACM/IEEE International Symposium on Networks-on-Chip, pp. 25–32. NOCS 2011, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1999946.1999951
24. Verbeek, F., Schmaltz, J.: Easy formal specification and validation of unbounded Networks-on-Chips architectures. ACM Trans. Des. Autom. Electron. Syst. **17**(1) (2012). https://doi.org/10.1145/2071356.2071357
25. Wu, J., Zhang, Z., Myers, C.: 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)
26. Yoneda, T., et al.: Network-on-Chip based multiple-core centralized ECUs for safety-critical automotive applications. In: Asai, S. (ed.) VLSI Design and Test for Systems Dependability, pp. 607–633. Springer, Tokyo (2019). https://doi.org/10.1007/978-4-431-56594-9_19
27. Zaman, A., Hasan, O.: Formal verification of circuit-switched Network on chip (NoC) architectures using SPIN. In: 2014 International Symposium on System-on-Chip, SoC 2014. Institute of Electrical and Electronics Engineers Inc. (December 2014). https://doi.org/10.1109/ISSOC.2014.6972449
28. Zhang, Z., Serwe, W., Wu, J., Yoneda, T., Zheng, H., Myers, C.: Formal analysis of a fault-tolerant routing algorithm for a Network-on-Chip. In: Lang, F., Flammini, F. (eds.) FMICS 2014. LNCS, vol. 8718, pp. 48–62. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10702-8_4
29. Zhang, Z., Serwe, W., Wu, J., Yoneda, T., Zheng, H., Myers, C.: An improved fault-tolerant routing algorithm for a Network-on-Chip derived with formal analysis. Sci. Comput. Program. **118**, 24–39 (2016). https://doi.org/10.1016/j.scico.2016.01.002, http://www.sciencedirect.com/science/article/pii/S0167642316000125, Formal Methods for Industrial Critical Systems (FMICS 2014)