Probabilistic Verification for Modular Network-on-Chip Systems*

Nick Waddoups[1], Jonah Boe[2], Arnd Hartmanns[3], Prabal Basu[4], Sanghamitra Roy[1], Koushik Chakraborty[1], and Zhen Zhang[1]

[1] Utah State University, Logan, UT, USA
{nick.waddoups, sanghamitra.roy, koushik.chakraborty, zhen.zhang}@usu.edu
[2] Hill Air Force Base, Davis County, UT, USA
[3] University of Twente, Enschede, The Netherlands
a.hartmanns@utwente.nl
[4] Cadence Design Systems, San Jose, CA, USA

Abstract. Quantitative verification can provide deep insights into reliable Network-On-Chip (NoC) designs. It is critical to understanding and mitigating operational issues caused by power supply noise (PSN) early in the design process: fluctuations in network traffic in modern NoC designs cause dramatic variations in power delivery across the network, leading to unreliability and errors in data transfers. Further complicating these challenges, NoC designs vary widely in size, usage, and implementation. This case study paper presents a principled, systematic, and modular NoC modeling approach using the Modest language that closely reflects the standard hierarchical design approach in digital systems. Using the Modest Toolset, functional and quantitative correctness was established for several NoC models, all of which were instantiated from a generic modular router model. Specifically, this work verifies the functional correctness of a generic router, inter-router communication, and the entire NoC. Statistical model checking was used to verify PSN-related properties for NoCs of size up to 8×8.

1 Introduction

As modern digital systems grow in complexity, there is a need for efficient on-chip communications as the traditional shared data bus becomes overloaded when more sub-systems are added to a chip. *Network-on-Chip* (NoC) designs are a widely-used alternative in computer chips that improve communication throughput and efficiency of a system by allowing more parallel communication.

* N.W. and Z.Z. gratefully acknowledge the support of the ECE Department at Utah State University. A.H. was supported by the EU's Horizon 2020 R&I programme under MSCA grant 101008233 (MISSION), the Interreg North Sea project STORM_SAFE, and NWO VIDI grant VI.Vidi.223.110 (TruSTy). S.R. and K.C. were supported in part by National Science Foundation (NSF) grant CNS-2106237. 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 NSF.

NoCs have attracted substantial research attention for improving both reliability and performance, especially in complex designs such as System-on-Chips (e.g., [32,48]). As modern semiconductor production integrates more subsystems on a single chip, from standard processing units to specialized hardware accelerators [40], NoCs are increasingly being adopted. However, their unique design challenges require correctness guarantees, particularly in safety-critical systems where faults can cause catastrophic failure [6,26].

One major challenge in NoC designs is *power supply noise* (PSN), which occurs with sudden shifts in traffic volume across the network. These shifts lead to fluctuations in the power drawn by each router and can result in data corruption or loss. Previous work [5] shows that PSN is magnified by the shift to smaller chip technology, with PSN increasing from 40% of the supply voltage to around 80% when moving an 8×8 NoC from 32 nm to 14 nm technologies.

Computer chips are designed in stages, starting with an architectural level specification, then moving onto an implementation in a hardware description language, and finishing with a physical layout before manufacturing. Design changes later in the cycle are significantly more costly than they would be if made earlier in the cycle. Incorporating probabilistic verification during the architectural stage of the NoC design flow allows for *early* quantification of PSN in order to mitigate issues. While formal verification of NoCs is an active area of study, there is only limited work on probabilistic verification of small-scale NoCs [38,34]. We argue that probabilistic verification is *required* to thoroughly check for PSN-related issues, as they are highly dependent on the inherent stochastic nature of network packet generation patterns.

Building on the second author's M.S. thesis work [7], this paper has the following major contributions:

1. A highly modular NoC model, written in the probabilistic modeling language Modest [17,8], to allow easy construction of arbitrary-size NoCs and characterization of PSN (Section 4). Previous work [38] presents only probabilistic verification of a small-scale 2×2 NoC, modeled in a monolithic, non-modular manner, which does not scale. The parameterizable nature of our modular model enables conventional and probabilistic model checking of diverse NoC implementations.
2. Verification of functional correctness of a generic router, inter-router communication, and full NoC model (Section 5).
3. Comparative model-checking: By reconfiguring our modular 2×2 NoC model into one equivalent to the earlier monolithic 2×2 NoC [38], we identified a deep-buried discrepancy in the modeling of packet consumption (Section 7). After resolving the discrepancies, our model successfully reproduced the results from [38], validating our unique modular modeling approach.
4. Probabilistic verification of PSN-related properties for NoCs of size up to 8×8, demonstrating the scalability and flexibility of the modular approach, while uncovering PSN characteristics not captured in previous work [38].

5. An analysis of the impacts of routing algorithms, demonstrating the frameworks ability to characterize PSN effects and reveal routing and scheduling strategies to improve reliability early in the design process.

2 Related Work

Probabilistic Modeling Checking. Research and development of probabilistic verification tools has been carried out for decades, stemming from research in the 1980s [27]. Modern research has produced probabilistic verification tools such as MODEST [20], PRISM [29], STORM [23], among others [24,39,18,44]. They have been widely applied to fields from flight controllers [51] to hardware security and reliability [37,15,36,28] to synthetic biology [46,10,35,31]. The MODEST TOOLSET can analyze models with quantitative data [20] and includes a built-in statistical model checker [9] used to generate PSN probabilities in this work.

NoC Verification. Previous NoC verification work mainly focuses on functional correctness of NoCs. It has been used to verify the correct routing behavior [47,41], check security properties [42], and evaluate performance [1]. These works have led to a *qualitative* understanding of NoCs, but cannot fully quantify critical properties involving reliability.

Probabilistic NoCs in Modest. Previous probabilistic verification for NoCs [38,34] was performed using MODEST. [38] developed a 2×2 NoC to quantify the effects of PSN using probabilistic model checking. This model was developed as a proof of concept, but was modeled in a *monolithic* fashion where all behavior and state variables were explicitly declared. This style of modeling is not scalable, and mistakes are easy to make. The goal of this work is to significantly enhance the scalability and usability of the concepts in [38] to characterize PSN in NoCs.

3 Preliminaries

3.1 NoC Architecture

The NoC architecture presented in this paper is a variable-size square mesh network composed of individual routers. Figure 1a shows a 2×2 NoC composed of four individual routers and Figure 1b shows an individual router.

A 2×2 NoC is made up of four routers in a grid pattern. Each router is responsible for receiving packets and routing them to their destination. Routers are denoted using r_i, where i is the router ID.

A router contains first-in-first-out (FIFO) fixed-size *buffers* to store incoming network packets and *channels* to send packets. Buffers are represented by small blocks ("□□□") and channels by arrows ("→") in Figure 1. Additionally, "×" indicates that the buffer and channel on that side of the router are not connected which occurs when a router is positioned at the edge of the network.

A specific buffer in a router is addressed as r_i^b, where b is the buffer label and i is the router ID. The buffer is described either by its position relative to a router, such as r_i^{North}, or generally, such as r_i^{Source}. In digital design, a fixed-size FIFO buffer has queue-like semantics where elements are inserted in the back and popped from the front. In our models, a channel represents physical wires that carry network data from one router to another.

Packets are used to communicate data between routers in a NoC, similar to packets in a traditional computer network. Starting at one router, a packet is moved from router to router through the network until it reaches its destination. Packets are introduced into and removed from the NoC by a *processing element* (PE) connected to each router via the local buffer and channel. A PE could be a CPU, accelerator core, memory cache, or other digital circuit. To move across the network, packets are (1) popped from the front of a buffer by a router, (2) routed through a channel towards their destination, and (3) pushed into the buffer connected to the channel.

As a digital circuit, NoCs are synchronized by a global clock. In our model, each channel can only be used once per clock cycle to send a packet. Additionally, in our model, each buffer can only accept one new packet per clock cycle. The sending of packets happens simultaneously (i.e., during the same clock cycle) between all NoC routers.

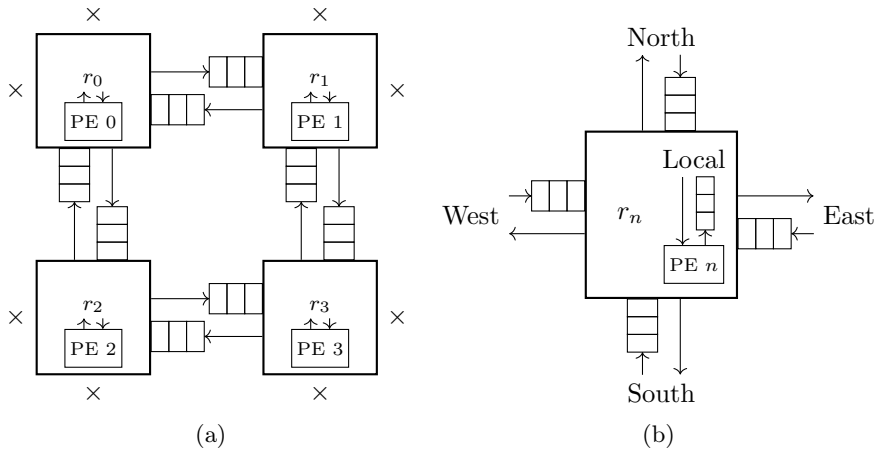


Fig. 1. A 2×2 NoC architecture and an individual router

In digital design, a packet is often split up into smaller components known as *flits* [5] due to limitations of channel bit-width. When a packet is transferred from one router to another, it is broken up into flits that are transmitted contiguously and sequentially over a channel, with one flit transferred per clock cycle. The number of flits in a packet is determined by the bit width of the channel, the size of the packet in bits, and the protocol being used.

The NoC model in this paper assumes a *single-flit packet* that can be transferred between routers in one clock cycle for the following reasons:

1. This work is targeted at characterizing PSN *early* in the design cycle, before the implementation of the NoC is complete. As such, the channel width and packet size may not be known when this modeling is intended to take place.
2. This work focuses on characterizing PSN based on high-level traffic patterns, routing algorithms, and NoC size. As such, our main consideration is to look at the high-level of packet traffic, not the individual movements of flits in the system. The abstraction presented in this work forgoes verification of circuit specific details for scalability and flexibility, which is desirable early in the design cycle. For more in-depth methods that look at characterizing PSN later in the design cycle, see [5].
3. Using single-flit packets greatly improves the tractability of the state space for verification. Roughly, a single-flit packet model has a state space complexity of $O(S^n)$, where S is the number of states for each buffer, and n is the number of buffers. Modeling multi-flit packets introduces many other possible states for each buffer, which would worsen the state space explosion problem and render verification intractable for the model.

Because we assume a single-flit packet, *packets* and *flits* are equivalent in our model as they each carry only a small piece of data and can be transferred from router to router in one clock cycle.

3.2 Routing Algorithm

Similar to computer networking, the path a flit takes to get from the sender to the receiver depends on the implemented routing protocol. The NoC design presented in this work allows *any* NoC routing algorithm to be modeled.

Results shown in this paper use the *X-Y routing algorithm*: It moves flits first horizontally across the NoC until they are in the same column as the destination router, and then moves them vertically until they reach their destination. This routing algorithm was chosen because it is deterministic, deadlock free, and is the same algorithm used in [38]. Additionally, the modularity of this work provides a framework for implementing other routing algorithms. More details about our implementation of the X-Y routing algorithm are given in Appendix A.4.

3.3 Router Arbitration

An important aspect of the router is the order in which each router's buffers are *serviced*, i.e. when a flit is removed from a buffer and sent through a channel to a destination. This order determines a router's fairness and is also critical to ensuring correct operation. An incorrect algorithm may deadlock, starve a particular router, or allow multiple flits to be routed per clock cycle across a single channel. We discuss routing correctness in Section 4.

Our NoC modeling framework uses CTL formulas of the form "forall globally" ($\forall\Box$) and "exists eventually" ($\exists\Diamond$) to ensure the functional correctness of a single router model and the composition to form an $n \times n$ NoC. *Probabilistic CTL* (PCTL) [19] is used to quantitatively check PSN-related properties. The PCTL syntax relevant to this work is $\mathbf{P}_{\sim q}(\Diamond^{\leqslant K}\Psi)$. That is, whether the probability that the state formula Ψ is eventually true within K steps lies within the interval specified by $\sim q$, where $\sim \in \{<, \leqslant, \geqslant, >\}$ and $q \in [0, 1]$. Additionally, the probabilistic operator $\mathbf{P}_{\sim q}$ can be replaced by $\mathbf{P}_{=?}$, indicating that the actual probability of the path formula $\Diamond^{\leq K}\Psi$ is of interest. For a complete definition of PCTL's syntax and semantics, see [30].

3.4 Power Supply Noise (PSN)

PSN has two components in a NoC: *resistive noise* – the product of the inherent resistance R of a circuit and the current I drawn, and *inductive noise* – the rate of change of current $\frac{\Delta I}{\Delta t}$ through the inherent inductance L of the circuit. The voltage drop due to PSN can radically degrade the delay of various on-chip circuit components, causing *timing errors*[5] in the network. Existing approaches to mitigate PSN are a far cry from a truly reliable NoC design paradigm, due to the lack of worst-case peak PSN guarantees [5,43].

Instead of directly measuring I, the model measures the activity of the router. Router activity is determined by the number of packets a router moves during a single clock cycle, ranging from zero (no packets) to five (all buffers used). This abstraction is supported by [5], which shows that high activity correlates to higher current, and an abrupt change in router activity leads to a high rate of current change. Like [38,34], PSN is tracked over clock cycles by introducing two counters, *resistiveNoise* and *inductiveNoise*. These counters represent the number of clock cycles where the router activity is over a user-specified router activity threshold T, where $T \in [0, 5]$.

Property 1 below is used to check the probability that the *resistiveNoise* counter will eventually exceed some threshold K within N clock cycles. Property 2 shows a similar property for determining the *inductiveNoise* probability within N clock cycles. Note that clock cycle progress is a *reward annotation* to certain transitions, i.e., accumulate(clk), instead of being encoded in the structure of an expanded state space.

$$\mathbf{P}_{=?}(\Diamond^{[\text{accumulate}(clk)\leqslant N]}\ resistiveNoise \geqslant K) \tag{1}$$

$$\mathbf{P}_{=?}(\Diamond^{[\text{accumulate}(clk)\leqslant N]}\ inductiveNoise \geqslant K) \tag{2}$$

This work characterizes PSN using the same method as [34,38]. This method abstracts away circuit-specific details in order to characterize PSN at a behavioral level. This is done because (1) early in the design process, engineers are

[5] A timing error is a fault caused when a pipe stage delay exceeds the clock period, leading to an erroneous value appearing in the network. In practice, this would appear as corrupted, dropped, or spurious data.

often more concerned with high-level behavior, (2) circuit-level netlist models are not available early in the design process, and (3) once made available, circuit-level netlist models are intractable for formal verification. This abstraction to a behavioral characterization of PSN in NoCs cannot be directly converted or compared to measured PSN on a digital circuit. However, the correlation between router activity and PSN shown in [5] indicates that the high-level abstraction used in this work enables valid characterization of PSN early in the design stage, providing key insights into the expected profile of PSN in the design.

3.5 Statistical Model Checking (SMC)

SMC is a quantitative verification approach that enables verifying the probability that a system model satisfies some property of interest. SMC uses data collected from random simulation runs to approximate the probabilistic properties of a model [33]. As a result, the confidence level of these approximations depends on the number of runs executed, with more executions resulting in higher confidence. Models containing rare events may take hundreds of thousands of runs to achieve accurate results. This work uses the SMC tool `modes` [9], part of the MODEST TOOLSET [20]. While the MODEST TOOLSET also contains tools for probabilistic model checking [2] (which uses exhaustive state space exploration), this work primarily uses SMC for its unparalleled scalability.

3.6 The Modest Toolset

The presented NoC architecture, routing algorithms, and properties are modeled and specified using the MODEST formal modeling language. Properties are then checked with the MODEST TOOLSET. While a comprehensive description of MODEST is outside the scope of this work, some concepts critical to the function of the model are outlined here. The `process` construct in MODEST models an asynchronous process and may contain local variables, atomic assignments, and synchronizing actions. Processes can be composed using either *sequential* or *parallel* composition. Parallel processes are synchronized through *actions*, which act as a synchronization barrier in a parallel composition where all participating processes must synchronize before performing the next step. A short example of `process` and composition syntax is given in Code Segment 1. We refer the interested reader to [22] for a tutorial and reference for MODEST.

The semantics of MODEST consists of two components: (1) a *symbolic semantics* [8, Sect. 4.2] that maps syntax to a symbolic *stochastic timed automaton* (STA) and (2) a *concrete semantics* [8, Sect. 5] that transforms the STA into a *timed probabilistic transition system* (TPTS), a form of uncountable-state Markov decision process. In our work, we use a subset of MODEST where the concrete semantics yields a *discrete-time Markov chain* (DTMC).

3.7 Rationale for Choosing MODEST

MODEST is our preferred modeling language and toolset for this work, as it is the *only* mature probabilistic formal modeling language that supports flexible data

structures for modular modeling, such as arrays, linked lists, and structs. These data structures are *not* supported by the other mature probabilistic modeling language, that of PRISM [29], yet they are key to our goal of developing a *highly modular* NoC model.

While the NoC model analyzed in this work is synchronous, MODEST and PRISM are inherently asynchronous. To resolve this, we can use the synchronizing actions offered by both MODEST and PRISM that force specified parts of the model to be synchronous. Other naturally synchronous formal modeling languages such as nuXmv [12] or Lustre [11,13] lack the required support for specifying models with probabilistic characteristics.

Finally, the MODEST TOOLSET has been actively maintained for many years and provides a suite of tools for statistical and probabilistic model checking of PCTL properties, with recently added support for checking CTL properties.

```
process P0() { /*...*/ } // process declaration
par { :: P0() :: P1() } // parallel composition
P0(); P1(); // sequential composition
```

Code Segment 1. Process and Parallel Composition Syntax in MODEST

4 Modular Design of the Formal NoC Model

We provide a framework for creating arbitrary $n \times n$ two-dimensional mesh NoCs in the MODEST language. The NoC model is a composition of instances of a router process, modeled using the `process` construct in MODEST. This modular style was influenced by modern digital design using *hardware description languages* such as Verilog [45] or VHDL [49], where modular design is essential for scalable designs and commonly used across all hardware designs. A modular design is defined by the clean separation of circuitry between different modules, where each module has a well defined interface that can be connected to other modules. Our modular design of the NoC model provides scalable results beyond that of previous works and more closely aligns with digital design practices.

This work presents a NoC model with easy customization of design parameters, including topological size, flit injection pattern, buffer size, and routing algorithm. The PSN characterization and verification of functional correctness is specified as properties *independent* of the specific model allowing for comparison of PSN characterization between models. Parameters such as topological size, buffer size, and PSN thresholds can be easily updated without knowledge of the MODEST language. Customization to behavior (e.g. packet injection pattern, routing algorithm, or priority arbitration) require writing MODEST code.

The full specification of the NoC model is too complex to fit within the page limit of this work. For this purpose, Appendix A has been made available with more details, examples, and results that support this work.

4.1 NoC Construction using Parallel Composition

An $n \times n$ NoC is described in our framework as the parallel composition of n^2 router processes and a clock process. For example, the 2×2 NoC shown in Figure 1a is a parallel composition of four routers and a clock, i.e., $r_0 \parallel r_1 \parallel r_2 \parallel r_3 \parallel \mathit{Clock}$. Code Segment 2 shows the parallel composition in MODEST, where each router is given its unique ID as input parameter with ID $\in [0, n^2 - 1]$. The router process demonstrates how a modular design improves the scalability of formal NoC models, as constructing a NoC of arbitrary size is accomplished by instantiating more routers.

```
par { :: Clock() :: Router(0) :: Router(1) :: Router(2) :: Router(3)}
```

Code Segment 2. Creation of a 2×2 NoC in MODEST

4.2 Router Process

The `Router` process models the generic router model in Figure 1b. This router has four input buffers and four output channels for interfacing with neighboring routers and a local input buffer for introducing new packets. The local buffer is the only way new packets are introduced into the NoC. The router model can be instantiated many times with a unique ID (Code Segment 2).

Each router maintains variables that define the state of the NoC, such as the elements in each buffer, previously serviced buffers, and various boolean flags that indicate what actions a buffer can take. These variables are stored in a global array of router objects. When instantiating a router process, the given `id` acts as an index into the array of router objects. These router objects cannot be completely default initialized due to design choices of MODEST, but a script is provided to automate this process as described in the Appendix A.5.

The buffers are modeled as a functional programming language-style *list*. This implementation efficiently models a FIFO queue, and allows for buffer size to be easily specified by a single global constant.

As shown in Code Segment 3, the router process is divided into five major sub-processes: generating new flits for the local buffer, preparing the router for flit advancement, advancing flits to their respective output channels, updating the buffer scheduling priority, and updating the PSN variables. After the `nextClockCycle` action, the router process calls itself recursively, which models the repeated execution of the same five sub-processes in every clock cycle.

Although the modular style allows for scalability, code reuse, and customization, it also initially had synchronization issues between routers that were not present in the previous work [38] because [38] did not use parallel composition to model the NoC. The parallel composition shown in Code Segment 2 is asynchronous, which means that each router instance executes its actions at its own speed without waiting for other routers.

```
process Router(int id) {
  GenerateFlits(id); PrepRouter(id);
  AdvanceRouter(id); UpdatePiority(id);
  nextClockCycle; /* synchronizing clock action */
  Router(id) /* recursive call models next clock cycle */ }
```

Code Segment 3. `Router` Process in MODEST

Two key issues presented themselves when initial verification attempts were performed on the NoC. First, the asynchronous composition meant that there was no guarantee that all routers would be synchronized. Because of this, one router could get ahead of its neighbors, effectively allowing multiple steps to be taken per clock cycle, which is not possible in a synchronous digital system. This error was fixed in [7] by introducing a global synchronization action, similar to `nextClockCycle` in Code Segment 3.

In MODEST, synchronization actions act as a barrier that every process must reach before proceeding. When the `nextClockCycle` action is reached, every process must wait for every other process to also reach the `nextClockCycle` action before proceeding. This prevents routers from performing multiple steps per clock cycle, and ensures that per-cycle PSN results are correct.

Unfortunately, a single `nextClockCycle` action is not sufficient to prevent all synchronization errors in the router. While the `nextClockCycle` action synchronizes the `Clock` and `Router` processes, errors can still occur where one flit passes through multiple routers in one clock cycle. This is a write-before-read conflict, where one router may call `AdvanceRouter` before another has called `PrepRouter`, and is not a valid NoC behavior.

For example, if r_0 has a flit destined for r_3 in the 2×2 NoC depicted in Figure 1a, it should take a minimum of two clock cycles to reach its destination. On clock cycle 1, the flit is sent from r_0 to r_1. Then, on clock cycle 2, it is sent from r_1 to r_3. However, if r_0 executes `AdvanceRouter` before r_1 executes `PrepRouter`, the flit could travel from r_0 to r_1 to r_3 in a single clock cycle.

A second synchronization action `sync` was inserted between the `PrepRouter` and `AdvanceRouter` processes in [7] and this successfully prevented write-before-read conflicts. However, when performing functional verification on the resulting model, the state space would explode due to the need to store states resulting from interleavings of each router *independently* executing each sub-process. These interleavings were not desired as they would not occur in a synchronous digital system. In this work, we inserted *fine-grained synchronization actions* into every sub-process so that all routers in the parallel composition executed in lock step. This model update not only alleviates state space explosion, but also more faithfully models synchronous digital hardware, where assignments across all components execute in lock step during every clock cycle.

4.3 Synchronizing Actions

The MODEST language is inherently asynchronous while most digital systems are synchronous with a global clock synchronizing all the updates in lock-step. To ensure that our MODEST model correctly models a synchronous system, we carefully employ fine-grained synchronization actions among all processes. In MODEST, each assignment block – assignments within `{= =}` braces – executes atomically. As described in this section, the NoC model is broken up into many processes, where each process handles one element of the model. Parallel composition of these processes produces one single model that acts as a NoC. In parallel assignment in MODEST, each unique interleaving of assignment blocks is explored. This produces many interleavings of actions, which is useful for many problems that MODEST can model. However, when modeling synchronous digital designs, we are not concerned with interleaving states, we are only concerned with the changes that occur with the synchronous clock.

Code segment 4 shows an example of a process that updates an element in array `x` based on the input `id`. The possible states including interleavings for these assignments are shown in Figure 2a. If this were modeling a digital circuit, and `x` was a digital register block that updated with the clock, states s_1 and s_2 would never be observed due to the synchronous semantics of the digital system, as shown in the clock diagram in Figure 2b. In practice, this problem of unnecessary interleaving causes an exponential explosion of state space that makes the modular model challenging to verify with the available hardware. This issue was not a problem for previous work [38] because that model was not made of modular processes composed together, but was rather represented by a large process that orchestrated each part of the model and updates occurred in the same assignment block.

```
int x[] = {0, 0};
process setTo5(int id) {{= x[id] = 5; =}}
par { :: setTo5(0) :: setTo5(1) }
```

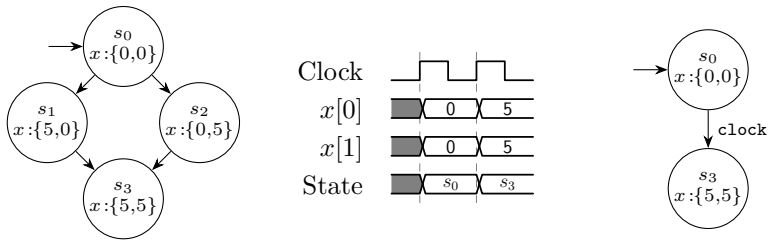
Code Segment 4. Parallel Interleaving Example

To resolve this issue we use the synchronizing actions provided by MODEST. Actions provide a synchronization method where parallel assignments happen in lockstep with one other. For a more in-depth explanation of actions in MODEST, see [22]. Adding a `clock` action that models the synchronous nature of digital circuits eliminates states s_1 and s_2, and we are left with a single state transition that acts akin to the synchronous digital system as shown in Figure 2c. An alternative would be to apply probabilistic partial order reduction [3,4,14] to pick *one interleaving* of the independent assignments; by exploiting the system's synchronous nature, we achieve a stronger reduction that collapses the entire interleaving into *one transition* synchronizing all the involved parallel processes.

To synchronize the parallel composition of router processes, each assignment in each sub-process called in `router` has a unique synchronizing action associated

```
action clock; int x[] = {0, 0};
process setTo5(int id) { clock{= x[id] = 5; =} }
par { :: setTo5(0) :: setTo5(1) }
```

Code Segment 5. Synchronizing Parallel Interleavings with a Clock Action



(a) Without actions (b) Digial semantics (c) With actions

Fig. 2. Adding Actions to Reduce Unnecessary State Space

to it. This ensures that in each state update the synchronous semantics of digital circuits is correctly represented and, like Figure 2 shows, helps to reduce the state space and improve verification time. Code Segment 3 shows the relevant sub-processes to be synchronized. Code Segment 6 shows an example of how each sub-process of `Router` uses a unique action to ensure that the assignments are synchronized. The effect of these actions is that each of the routers update their state together, and no interleaving is allowed. Figure 3 shows an example trace of the state updates of an $n \times n$ NoC, where each router updates in sync with the other routers.

```
action gf, pr, ar, up;
process GenerateFlits(int id) { gf {= /*...*/ =} }
process PrepRouter(int id)    { pr {= /*...*/ =} }
process AdvanceRouter(int id) { ar {= /*...*/ =} }
process UpdatePiority(int id) { up {= /*...*/ =} }
```

Code Segment 6. Use of Synchronizing Actions in Router Processes

4.4 Clock Process

A clock process (Code Segment 7) is composed with the router instances to count the elapsed clock cycles (Code Segment 2). The clock count is used to generate the PSN results (Section 8) and to give context to different flit injection strategies. MODEST allows for variable assignment to occur with a synchronization action (i.e. `nextClockCycle`), which is used to increment the clock count.

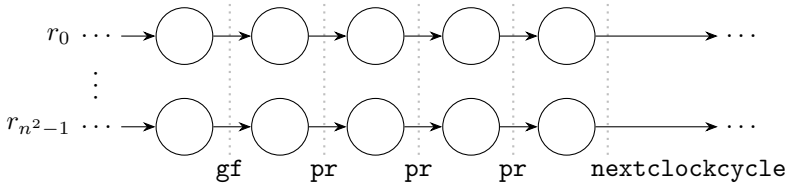


Fig. 3. Impact of Synchronizing Actions in a Parallel Composition of n^2 Routers

```
process Clock() { nextClockCycle{= clk = (clk + 1) =}; Clock()}
```

Code Segment 7. `Clock` Process in MODEST

4.5 Flit Generation Algorithm

The modular `GenerateFlits` process allows for easy customization of network traffic patterns in order to measure the PSN impact of different flit injection strategies. This process has one input parameter `id` and may add a flit to r_{id}^{L}. This simulates the generation of network traffic in the NoC. Determining the traffic generation pattern needs to be done after observing patterns in real-world or simulated environments, as the generation pattern will vary based on application and use-case.

Results in this work use the periodic flit generation pattern from [38] with a 3/10 duty cycle. In this pattern, each router has a new flit synchronously added to the local buffer during the first 3 out of every 10 clock cycles. Destinations for each new flit are chosen according to a uniform distribution of all other routers in the NoC. While this specific distribution and generation pattern were chosen for direct comparison with [38], the modular framework allows arbitrary generation patterns and destination distributions. Appendix A.4 gives further examples.

4.6 Routing Algorithm

The *routing algorithm* used by each router determines how incoming flits should be treated. This is another modular process with input parameter `id` that can be adjusted to model different routing algorithms in the NoC. Due to the page limit, the full details and semantics of the `AdvanceRouter` process are included in the Appendix A.4.

This work implements the X-Y routing algorithm, as described in Section 3, because it is deterministic and deadlock free. X-Y routing was used in [38,34] and a goal of this work was to reproduce results from those papers. Additionally, 7 of the 16 real-world NoCs surveyed in [25] used X-Y (or Y-X) routing.

4.7 Priority Tracking and Updates

During routing, flits are passed between buffers on channels. Since a channel can be used *only once* per clock cycle, conflicts may arise when multiple flits need to use the same channel. To ensure that all buffers are *serviced*, flags are set when

a buffer cannot be serviced due to conflicts. A buffer is *serviced* when a flit is removed from the front and removed or sent down a channel to another buffer of the neighboring router.

The `UpdatePriority` sub-process uses these flags to generate the priority of buffers to service for the next cycle. The order of buffers to be serviced next is stored in an array and used during the next clock cycle as the initial order for flits to be sent. As part of the modularity of this design, the priority algorithm used for each buffer can be reconfigured in the model to meet the design goals. This work uses a round-robin process to ensure that each buffer is serviced fairly.

4.8 Noise Tracking

PSN is characterized by tracking the activity level of each router in the network every clock cycle. As discussed in Section 3.4, resistive noise is related to the activity during the current clock cycle, and inductive noise is related to the change in activity over two consecutive clock cycles. Further discussion of PSN characterization is given in Section 6.

5 Correctness of the Modular NoC

MODEST uses the NoC model we provide to explore the complete state-space, and then the MODEST TOOLSET's `mcsta` model checker [21] uses $\forall\Box$ and $\exists\Diamond$ CTL formulas to verify the correctness of a NoC design. Verification of these properties is carried out on a generic router and inter-router communications. In this section, i and j represent valid router IDs in a NoC and $j \twoheadrightarrow r_i^L$ denotes a packet destined for j being added to the *Local* buffer of r_i.

5.1 Flit Generation Verification

As discussed in Section 4.5, the flit generation pattern for a given NoC design can be easily modified. Thus verifying a unique flit generation pattern requires a verification strategy suited to each pattern, so we do not present a general method for verifying the correctness of *all* flit generation patterns.

However, one property that all flit generation patterns must demonstrate across all paths is that a flit destined for the router they are originating from should not be generated as shown in Property 3.

$$\forall i : \forall\Box(\neg(i \twoheadrightarrow r_i^L)) \tag{3}$$

While we do not present a method for verifying all possible flit generation patterns, we do demonstrate a possible approach to such verification. The flit generation algorithm used in [38] assigns destinations using a uniformly random distribution. We can verify that there exists an execution path in which a flit is sent from router i to router j for all routers i, j using the following CTL property:

$$\forall i : \forall j : j \neq i \implies \exists\Diamond(j \twoheadrightarrow r_i^L) \tag{4}$$

5.2 Scheduling Priority Verification

The implementation of this scheduler relies on a list of buffers that have been serviced or are waiting for service. It is important that this list is coherent, i.e. that the list always contains all buffers and no duplicate entries. Property 5 shows the CTL formula for this property, where i is a placeholder for router ID.

$$\begin{aligned}\forall i : \forall\Box(&\mathit{local} \in r_i.\mathit{priorityList} \wedge \mathit{north} \in r_i.\mathit{priorityList} \\ &\wedge\ \mathit{east} \in r_i.\mathit{priorityList} \wedge \mathit{south} \in r_i.\mathit{priorityList} \\ &\wedge\ \mathit{west} \in r_i.\mathit{priorityList}) \wedge \text{length}(\mathit{priorityList}) = 5\end{aligned} \tag{5}$$

5.3 Flit Propagation Verification

The real-world NoC design deploys buffers of bounded sizes, but they are modeled as an unbounded queue in Modest. This design choice is used to facilitate the instantiation of a buffer of arbitrary size, but makes it possible for a bounded buffer to exceed its specified size in the model. Therefore, we must specify a property to check that all the buffers in a NoC never exceed the user-specified size. This property should also check that a flit is never accidentally sent to a full buffer, as that would increase the length of the buffer by one. In CTL, this is

$$\forall i, j : \forall\Box(\text{length}(r_i.\mathit{buffer}_j) \leqslant \mathit{BUFFER\_SIZE}) \tag{6}$$

Additionally, each channel should only admit at most one flit during a clock cycle to correctly model a synchronous NoC. We verify this property by adding a counter to each channel that tracks how many times it was used during a single clock cycle. It is incremented when a flit is sent across the channel during the `AdvanceRouter` process, and is reset back to zero in the `UpdatePriority` process. The CTL property to check that no counter is ever greater than one is

$$\forall i, j : \forall\Box(r_i.\mathit{channel}_j.\mathit{used\_count} \leqslant 1) \tag{7}$$

5.4 Improvement Over Previous Works

Previous works such as [38,7] were not able to perform CTL model checking as `mcsta` did not support CTL then. Instead, they encoded functional correctness using the PCTL syntax and checked if the probability of PCTL encodings of Properties 3-7 were 0.0 or 1.0, which may not be the same for certain probabilistic loops, and requires less scalable model checking methods.

Additionally, due to the many unnecessary interleavings generated by the lack of true synchronization as discussed in Section 4.2, previous models were only able to be checked up to a small number of clock cycles, effectively performing bounded model checking. Our work checks an unbounded number of cycles for the 2×2 NoC. Checking 3×3 however still runs out of memory; future work can look at abstracting parts of the modular model to scale up verification.

6 PSN Probability

As discussed in Section 3.4, this work measures PSN through an behavioral description instead of circuit level details such as resistance, inductance, and current. This section outlines our method for calculating PSN probabilities and how it is customized to provide more information.

Resistive noise occurs when activity occurs in a router during a single clock cycle. Each router maintains a local `activity` counter to track events that occur during a single clock cycle. The counter `activity` increases when a packet is sent or received. This activity level translates into a resistive noise event when it crosses the activity threshold K (i.e., `ACTIVITY_THRESH`) set by the user. We then use Property 1 to determine the cumulative probability that N resistive noise events occur in k clock cycles. N is set by the user as `RESISTIVE_THRESH`.

Inductive noise occurs when the activity level in a router changes between two clock cycles. Each router also maintains a local `lastActivity` variable that stores the activity of the previous clock cycle. When the difference between `activity` and `lastActivity` is greater than the activity threshold K, it generates an inductive noise event. We then use Property 2 to determine the cumulative probability that M inductive noise events occur. M is set by the user as `INDUCTIVE_THRESH`.

By default, the PSN characterization in this model is a global characterization, where PSN is considered for all routers in the network. However, since each router maintains a local `activity` and `lastActivity` variable, it is easy to extract PSN probabilities for a single router or group of routers instead of all routers, as shown in Section 7.

7 Results and Discussion

The results presented in this work were generated on a machine using a 12-core AMD Ryzen Threadripper Processor (at 3.5 GHz), 132 GB of memory, and Ubuntu Linux version 22.04 LTS. MODEST TOOLSET version 3.1.290 was used. All formal models, plots, and scripts are publicly available at [16]. The runtime results are presented in Table 1.

7.1 CTL Verification Results for a 2×2 NoC

CTL model checking results for the 2×2 NoC were generated using `mcsta`, the model checking engine in MODEST. In addition, the `chainopt` option was used to reduce the state space by collapsing chains of states connected by probability-1 transitions. All properties specified in Section 5 were verified on the 2×2 model.

7.2 PSN Results

The results of the PSN PCTL properties were captured using `modes`, the statistical model checker in MODEST. SMC was used due to memory constraints for

Size	Parameter	Tool	Confidence Interval	Runtime (HH:MM:SS)	Memory (MB)	States
2×2	Functional Properties	`mcsta`	–	00:10:31	9844	3446073
2×2	Resistive PSN	`modes`	95%	00:06:11	105	–
2×2	Inductive PSN	`modes`	95%	00:56:08	105	–
3×3	Resistive PSN	`modes`	95%	00:03:07	129	–
3×3	Inductive PSN	`modes`	95%	00:14:13	131	–
4×4	Resistive PSN	`modes`	95%	00:02:42	164	–
4×4	Inductive PSN	`modes`	95%	00:16:13	172	–
8×8	Resistive PSN	`modes`	95%	00:05:23	545	–
8×8	Inductive PSN	`modes`	95%	00:21:08	565	–
Total				02:14:36		

Table 1. Runtime Results

models larger than 2×2. In `modes`, the `max-run-length` option is set to zero to allow longer simulation runs (as `modes` by default aborts when it encounters a simulation run longer than 10,000 transitions, as a heuristic to avoid hanging on unintended infinite paths – which our model does not have).

The main goal of this work was to create a modular NoC framework that is more flexible and scalable than the previous work. However, in order to reproduce and compare the probabilistic verification results between this work and similar previous work [38], we configure the NoC model to match the previous ones. Therefore, this model uses a buffer length of four, the X-Y routing algorithm, an activity threshold of three, and the 3/10 flit injection pattern (detailed in Section 4.5) unless otherwise stated.

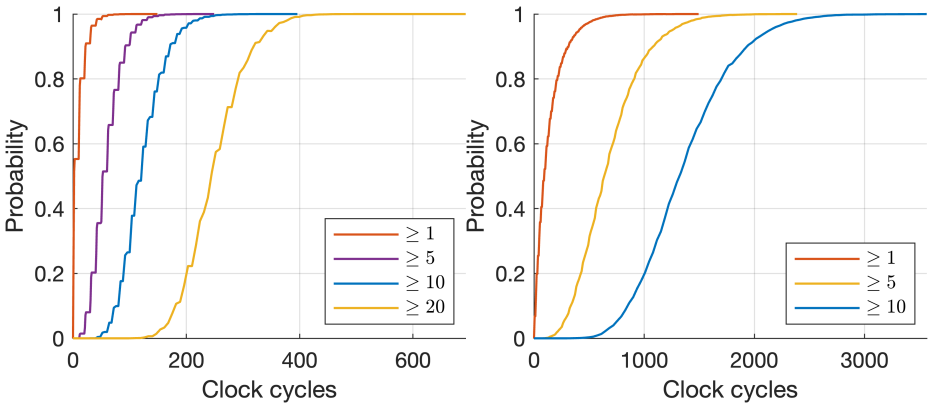
2×2 NoC Verification Results Figure 4 shows the PSN results for a 2×2 NoC for our modular design. Each plot shows the *cumulative distribution functions* (CDFs) of the probability of PSN events greater than or equal to the amount shown in the legend. Both resistive and inductive noise display a step-like growth corresponding to the 3/10 flit injection pattern detailed in Section 4.5.

7.3 Comparison to Previous 2 × 2 NoC Model

The 2 × 2 model results in [38] were regenerated using the `modes` SMC tool and compared the modular 2×2 results of this work in order to reproduce their findings. A similar comparison of a central router from [34] to the modular model is given in [7].

The initial results of the modular design were quite different from [38]. Because the modular model had been proved functionally correct by passing the CTL properties detailed in Section 5, we determined the discrepancies derived either from architectural differences or from an error in previous work.

We used comparative model checking to narrow down the differences between the two models and discovered two architectural differences. The initial modular



(a) Inductive Noise (b) Inductive Noise

Fig. 4. PSN for 2×2 Modular Model

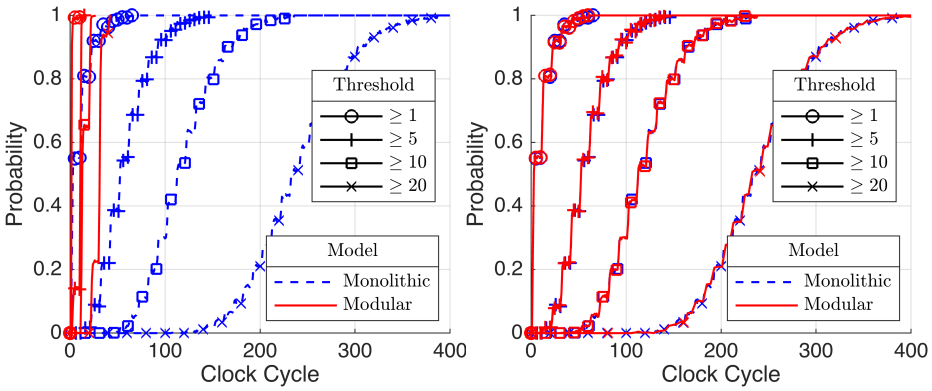
implementation did not generate new flits on clock cycle 0, while the previous monolithic model of [38] did. We corrected this by rearranging the sub-processes shown in Code Segment 3 to occur before the synchronizing clock action.

Additionally, the modular model allows multiple flits to be consumed during a single clock cycle, while the monolithic model allowed only a single flit to be consumed per cycle. For example, if two flits destined for r_0 arrived at r_0 on the same clock cycle in different channels, the modular model would consume both flits, while the monolithic model would consume only one. While both designs are valid, the reality is that the hardware attached to each router is unknown. It is therefore *assumed* in this work that the hardware attached to each router is capable of consuming multiple flits per cycle, and we adjusted the monolithic model to accept multiple flits per clock cycle.

We performed comparative model checking by assuming that both models were correct, and then generating the PSN characterization for each model and comparing the output. If the output differs, at least one of the models is faulty as they should be equivalent. Determining which model is faulty is done by simulating traces where the correct output is known on each model. Whichever model returns the incorrect output is the faulty model, and the trace can be analyzed to determine where the error originated. This process is repeated iteratively until the models match. We used this method because we could not directly compare states due to the different implementations of each model. Matching results after comparative-model checking was completed is shown in Figure 5.

7.4 PSN Results for Larger NoCs

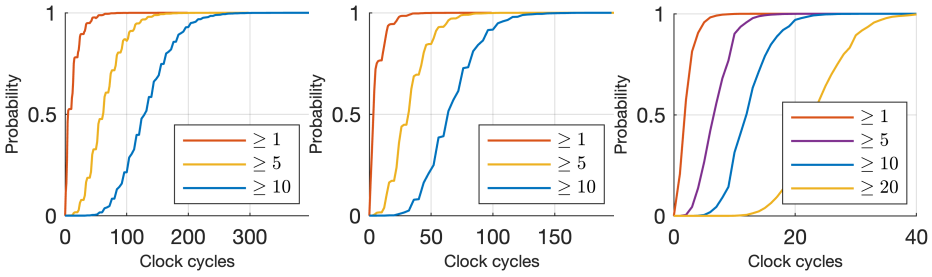
The advantage of a modular NoC model is the ability to scale it with ease. By instantiating 9, 16, and 64 routers for 3×3, 4×4, and 8×8 NoCs, respectively, we are quickly able to scale our PSN analysis. Figure 6 shows the inductive



(a) Before Comparative Model Checking (b) After Comparative Model Checking

Fig. 5. Modular and Monolithic 2×2 NoC Resistive Noise Comparison

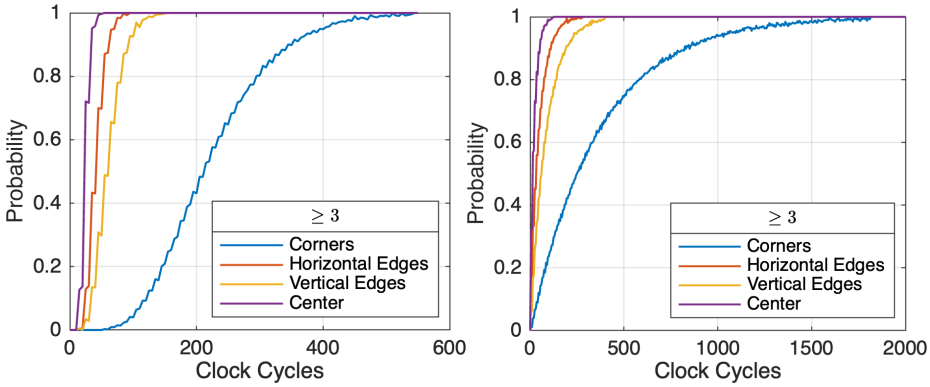
noise CDFs for these sizes. Resistive noise CDFs are not shown here due to page limitations, but are shown in Appendix A.1.



(a) 3×3 (b) 4×4 (c) 8×8

Fig. 6. Inductive Noise CDF for 3×3, 4×4, and 8×8 Modular Model

Similar to the 2×2 results in Figure 4, the 3×3 and 4×4 results show a pronounced pattern of steep slopes followed by rough edges which is a result of our chosen 3/10 flit injection pattern. As expected, compared to a 2×2 NoC, the likelihood of PSN events is much higher and occurs sooner in the 3×3 and 4×4 NoCs due to the higher number of routers and network traffic. Inductive noise events are likely to happen sooner in the 8×8 NoC compared to smaller models, and the pronounced steps every 3/10 cycles are gone. This is because packets are much more likely to persist in the 8×8 NoC, as packets in the 8×8 are likely to travel farther.



(a) Resistive Noise (b) Inductive Noise

Fig. 7. Router Specific PSN for 3×3 Modular Model

7.5 PSN for Specific Routers

One difference between a 2×2 and 3×3 NoC are the different routing connections used. A 2×2 NoC, such as the one shown in Figure 1a, only uses *corner routers* with two neighbors. A 3×3 configuration has *corner routers* with two neighbors, *edge routers* with three neighbors, and a *central router* with four neighbors.

The modular NoC model allows us to obtain insights that were impossible to obtain from the monolithic design style [38,34]. Rather than checking PSN globally, we can inspect the `activity` and `lastActivity` of an individual router to get a CDF for PSN for that specific router. The results of PSN resulted from different router types are shown in Figure 7. PSN results were generated for each router type with an `ACTIVITY_THRESH` of 3.

The router-specific results in Figure 7 show that the PSN increases toward the center of the network. Although the corner routers experienced a very gradual increase in PSN with respect to time, the center router experienced high levels of PSN almost immediately. This is because the central router has more active buffers than the others, and consequently, it experiences higher PSN. However, the horizontal edge routers experienced a higher PSN than those on the vertical edges, regardless of having the same number of buffers. This is a result of the X-Y routing causing more traffic through those areas. Therefore, placing most traffic-prone neighbors on the NoC perimeter may reduce PSN as the effects of PSN are less prevalent on corner and edge routers.

This ability to check PSN for specific parts of a NoC can be used to provide valuable information on how the chosen routing algorithm and the flit injection pattern affect PSN. For example, if a new routing algorithm was designed to minimize traffic to the central router of a 3×3 NoC, our modular model could verify the effects on PSN that the new algorithm would have compared to the current X-Y routing algorithm. This would allow routing algorithm design to occur early in the design cycle, leading to more effective designs.

Previously, the largest topology achieved for a probabilistic NoC model was a 2×2 network. As mentioned, 2×2 routers would only have three input buffers (two neighbor buffers and a local buffer). With the PSN threshold set to three, this inherently results in lower PSN between routers with higher conflict rates, because if even one buffer goes unserviced in a cycle, there can be no PSN on that router. PSN characterization for 2×2 networks does not scale to larger NoCs, making it difficult to make any particular recommendation to chip designers based on 2×2 findings alone.

In the work of [38], it is concluded that PSN can be reduced by injecting delays between flits. While this is true, this work affirms that PSN is difficult to completely eradicate using the flit injection pattern alone. The larger a NoC becomes, the longer it takes on average for flits to propagate through the system. Injecting flits in only a single clock cycle can still result in PSN. In addition to reducing the flit injection rate, this work recommends grouping routers into high-traffic regions, with the highest traffic connections reserved for vertical neighbors. In this way, flits can propagate through the NoC more quickly. This recommendation, in conjunction with keeping heavy traffic on the perimeter of the NoC, may result in even lower traffic through the central routers.

8 Conclusion

This paper presents a case study on the development of an easy-to-use, scalable, and modular formal NoC model using the MODEST modeling language. The paper describes the CTL properties to verify the correctness of the model and the PCTL properties to quantify PSN. It also describes the diagnoses of a discrepancy in the quantification of PSN in a 2×2 NoC while attempting to reproduce a previous work [38], which revealed an inaccurate modeling of the flit consumption behavior in [38]. In addition, results from statistical model checking on the 3×3, 4×4, and 8×8 NoCs suggested network flit scheduling schemes to minimize PSN. While this work focuses on NoCs sized up to 8×8, it can be easily scaled to larger models. The modular design of the model makes it ideal for examining NoC designs early in the design cycle. Doing so may help find potential errors sooner and may encourage designs that are more robust in their function. Future work includes scaling the CTL correctness verification to arbitrarily sized models and exploring a wide range of flit injection patterns and routing algorithms.

Data Availability Statement

A docker environment with the tools, models, and results of this paper is available on `Zenodo` [50]. Additionally, the models are available on `GitHub` [16].

References

1. Alhubail, L., Bagherzadeh, N.: Power and performance optimal noc design for cpu-gpu architecture using formal models. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 634–637 (2019). https://doi.org/10.23919/DATE.2019.8714769
2. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_28
3. Baier, C., D'Argenio, P.R., Größer, M.: Partial order reduction for probabilistic branching time. In: Cerone, A., Wiklicky, H. (eds.) Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005). Electronic Notes in Theoretical Computer Science, vol. 153, pp. 97–116. Elsevier (2005). https://doi.org/10.1016/J.ENTCS.2005.10.034
4. Baier, C., Größer, M., Ciesinski, F.: Partial order reduction for probabilistic systems. In: 1st International Conference on Quantitative Evaluation of Systems (QEST 2004). pp. 230–239. IEEE Computer Society (2004). https://doi.org/10.1109/QEST.2004.1348037
5. Basu, P., Shridevi, R.J., Chakraborty, K., Roy, S.: IcoNoClast: Tackling voltage noise in the NoC power supply through flow-control and routing algorithms. IEEE Trans. VLSI Syst. **25**(7), 2035–2044 (2017)
6. Becchu, N.K.R., Harishchandra, V.M., Yernad Balachandra, N.K.: System level fault-tolerance core mapping and fpga-based verification of noc. Microelectronics Journal **70**, 16–26 (2017). https://doi.org/https://doi.org/10.1016/j.mejo.2017.09.010, https://www.sciencedirect.com/science/article/pii/S0026269217302884
7. Boe, J.W.: Probabilistic Verification for Modular Network-on-Chip Systems. Master's thesis, Utah State University (2023). https://doi.org/10.26076/4f5a-4a68, https://digitalcommons.usu.edu/etd/8763
8. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. **32**(10), 812–830 (2006). https://doi.org/10.1109/TSE.2006.104
9. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: A statistical model checker for nondeterminism and rare events. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 340–358. Springer (2018)
10. Buecherl, L., Roberts, R., Fontanarrosa, P., Thomas, P.J., Mante, J., Zhang, Z., Myers, C.J.: Stochastic hazard analysis of genetic circuits in iBioSim and STAMINA. ACS Synthetic Biology **10**(10), 2532–2540 (2021). https://doi.org/10.1021/acssynbio.1c00159, https://doi.org/10.1021/acssynbio.1c00159, pMID: 34606710
11. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 178–188. POPL '87, Association for Computing Machinery, New York, NY, USA (1987). https://doi.org/10.1145/41625.41641, https://doi-org.dist.lib.usu.edu/10.1145/41625.41641
12. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In:

Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Kobsa, A., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Terzopoulos, D., Tygar, D., Weikum, G., Biere, A., Bloem, R. (eds.) Computer Aided Verification, vol. 8559, pp. 334–342. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22

13. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The kind 2 model checker. In: International Conference on Computer Aided Verification (2016), https://api.semanticscholar.org/CorpusID:11582048
14. D'Argenio, P.R., Niebert, P.: Partial order reduction on concurrent probabilistic programs. In: 1st International Conference on Quantitative Evaluation of Systems (QEST 2004). pp. 240–249. IEEE Computer Society (2004). https://doi.org/10.1109/QEST.2004.1348038
15. Fang, L., Yamagata, Y., Oiwa, Y.: Evaluation of A resilience embedded system using probabilistic model-checking. In: Pang, J., Liu, Y. (eds.) Proceedings Third International Workshop on Engineering Safety and Security Systems, ESSS 2014, Singapore, Singapore, 13 May 2014. EPTCS, vol. 150, pp. 35–49 (2014). https://doi.org/10.4204/EPTCS.150.4, https://doi.org/10.4204/EPTCS.150.4
16. Artifact repository for "probabilistic verification for modular network-on-chip systems" (Sep 2025), https://github.com/formal-verification-research/VMCAI26_Modular_NoC_Artifact, original-date: 2025-09-16T01:29:03Z
17. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013). https://doi.org/10.1007/S10703-012-0167-Z
18. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: A web-based probabilistic model checker. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods. pp. 312–317. Springer International Publishing, Cham (2014)
19. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6**(5), 512–535 (1994). https://doi.org/10.1007/BF01211866, https://doi.org/10.1007/BF01211866
20. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS. LNCS, vol. 8413, pp. 593–598. Springer (2014)
21. Hartmanns, A., Hermanns, H.: Explicit model checking of very large MDP using partitioning and secondary storage. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9364, pp. 131–147. Springer (2015). https://doi.org/10.1007/978-3-319-24953-7_10
22. Hartmanns, A., Hermanns, H.: A Modest Markov automata tutorial. In: Krötzsch, M., Stepanova, D. (eds.) Reasoning Web. Explainable Artificial Intelligence - 15th International Summer School 2019, Bolzano, Italy, September 20-24, 2019, Tutorial Lectures. Lecture Notes in Computer Science, vol. 11810, pp. 250–276. Springer (2019). https://doi.org/10.1007/978-3-030-31423-1_8, https://doi.org/10.1007/978-3-030-31423-1_8
23. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (aug 2022). https://doi.org/10.1007/s10009-021-00633-z, https://doi.org/10.1007/s10009-021-00633-z
24. Jeppson, J., Volk, M., Israelsen, B., Roberts, R., Williams, A., Buecherl, L., Myers, C.J., Zheng, H., Winstead, C., Zhang, Z.: STAMINA in C++: modernizing

an infinite-state probabilistic model checker. In: Jansen, N., Tribastone, M. (eds.) Quantitative Evaluation of Systems - 20th International Conference, QEST 2023, Antwerp, Belgium, September 20-22, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14287, pp. 101–109. Springer (2023). https://doi.org/10.1007/978-3-031-43835-6_7, https://doi.org/10.1007/978-3-031-43835-6_7
25. Jerger, N.E., Krishna, T., Peh, L.S.: On-Chip Networks. Synthesis Lectures on Computer Architecture, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-031-01755-1
26. Jiang, S.Y., Luo, G., Liu, Y., Jiang, S.S., Li, X.T.: Fault-tolerant routing algorithm simulation and hardware verification of noc. IEEE Transactions on Applied Superconductivity **24**(5), 1–5 (2014). https://doi.org/10.1109/TASC.2014.2346484
27. Katoen, J.P.: The probabilistic model checking landscape. In: 2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–15 (2016)
28. Kumar, J.A., Vasudevan, S.: Automatic compositional reasoning for probabilistic model checking of hardware designs. In: QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010. pp. 143–152. IEEE Computer Society (2010). https://doi.org/10.1109/QEST.2010.25, https://doi.org/10.1109/QEST.2010.25
29. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)
30. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking, pp. 220–270. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72522-0_6, https://doi.org/10.1007/978-3-540-72522-0_6
31. Lakin, M., Parker, D amd Cardelli, L., Kwiatkowska, M., Phillips, A.: Design and analysis of dna strand displacement devices using probabilistic model checking. In: Journal of the Royal Society, Interface Vol. 9. Journal of the Royal Society, Interface (jan 2012). https://doi.org/10.1098/rsif.2011.0800
32. Lecler, J.J., Baillieu, G.: Application driven network-on-chip architecture exploration and refinement for a complex soc. Design Autom. for Emb. Sys. **15**, 133–158 (06 2011). https://doi.org/10.1007/s10617-011-9075-5
33. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. pp. 122–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
34. Lewis, B., Hartmanns, A., Basu, P., Jayashankara Shridevi, R., Chakraborty, K., Roy, S., Zhang, Z.: Probabilistic verification for reliable network-on-chip system design. In: Larsen, K.G., Willemse, T. (eds.) Formal Methods for Industrial Critical Systems. pp. 110–126. Springer International Publishing, Cham (2019)
35. Madsen, C., Myers, C.J., Roehner, N., Winstead, C., Zhang, Z.: Utilizing stochastic model checking to analyze genetic circuits. In: 2012 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB). pp. 379–386 (5 2012). https://doi.org/10.1109/CIBCB.2012.6217255
36. Maes, R.: An accurate probabilistic reliability model for silicon pufs. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 73–89. Springer (2013)
37. Mundhenk, P., Steinhorst, S., Lukasiewycz, M., Fahmy, S.A., Chakraborty, S.: Security analysis of automotive architectures using probabilistic model checking. In: Proceedings of the 52nd Annual Design Automation Conference. p. 38. ACM (2015)

38. Roberts, R., Lewis, B., Hartmanns, A., Basu, P., Roy, S., Chakraborty, K., Zhang, Z.: Probabilistic verification for reliability of a two-by-two network-on-chip system. In: Lluch Lafuente, A., Mavridou, A. (eds.) Formal Methods for Industrial Critical Systems. pp. 232–248. Springer International Publishing, Cham (2021)
39. Roberts, R., Neupane, T., Buecherl, L., Myers, C.J., Zhang, Z.: STAMINA 2.0: Improving scalability of infinite-state stochastic model checking. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 319–331. Springer International Publishing, Cham (2022)
40. Royannez, P., Mair, H., Dahan, F., Wagner, M., Streeter, M., Bouetel, L., Blasquez, J., Clasen, H., Semino, G., Dong, J., Scott, D., Pitts, B., Raibaut, C., Ko, U.: 90nm low leakage soc design techniques for wireless applications. In: ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005. pp. 138–589 Vol. 1 (2005). https://doi.org/10.1109/ISSCC.2005.1493907
41. Salamat, R., Khayambashi, M., Ebrahimi, M., Bagherzadeh, N.: A resilient routing algorithm with formal reliability analysis for partially connected 3d-nocs. IEEE Transactions on Computers **65**(11), 3265–3279 (2016). https://doi.org/10.1109/TC.2016.2532871
42. Sepulveda, J., Aboul-Hassan, D., Sigl, G., Becker, B., Sauer, M.: Towards the formal verification of security properties of a network-on-chip router. In: 2018 IEEE 23rd European Test Symposium (ETS). pp. 1–6 (2018). https://doi.org/10.1109/ETS.2018.8400692
43. Shridevi, R.J., Ancajas, D.M., Chakraborty, K., Roy, S.: Tackling voltage emergencies in noc through timing error resilience. In: ISLPED. pp. 104–109 (2015)
44. Song, S., Sun, J., Liu, Y., Dong, J.S.: A model checker for hierarchical probabilistic real-time systems. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 705–711. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
45. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language (2018). https://doi.org/10.1109/IEEESTD.2018.8299595
46. Taylor, L., Israelsen, B., Zhang, Z.: Cycle and commute: Rare-event probability verification for chemical reaction networks. In: Nadel, A., Rozier, K.Y. (eds.) Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design – FMCAD 2023. pp. 284–293. TU Wien Academic Press (2023). https://doi.org/10.34727/2023/isbn.978-3-85448-060-0{_}37
47. Taylor, L., Zhang, Z.: Scaling up livelock verification for network-on-chip routing algorithms. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 378–399. Springer International Publishing, Cham (2022)
48. Tsai, Lan, Hu, Chen: Networks on chips: Structure and design methodologies. Journal of Electrical and Computer Engineering **2012** (2012). https://doi.org/10.1155/2012/509465
49. IEEE Standard for VHDL Language Reference Manual (2019). https://doi.org/10.1109/IEEESTD.2019.8938196
50. Waddoups, N., Boe, J., Hartmanns, A., Basu, P., Roy, S., Chakraborty, K., Zhang, Z.: Artifact for "probabilistic verification for modular network-on-chip systems artifact" submitted to vmcai26 (Oct 2025). https://doi.org/10.5281/zenodo.17247418, https://doi.org/10.5281/zenodo.17247418
51. Wang, L., Cai, F.: Reliability analysis for flight control systems using probabilistic model checking. In: 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS). pp. 161–164 (2017). https://doi.org/10.1109/ICSESS.2017.8342887

Appendix

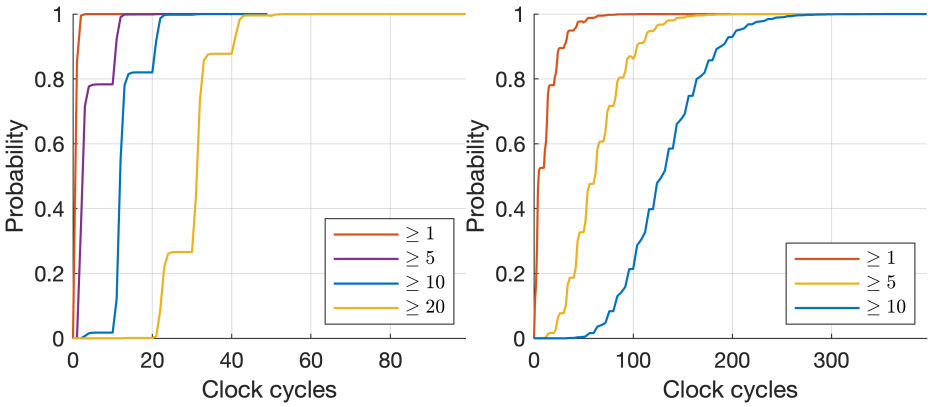
A Modular NoC Implementation

This section gives an outline of the NoC design implemented in MODEST. For information on how to generate unique models, see Appendix A.5, and for details on the each process in the model see Appendix A.4. Additional results are shown in Appendix A.1.

A.1 Additional Results

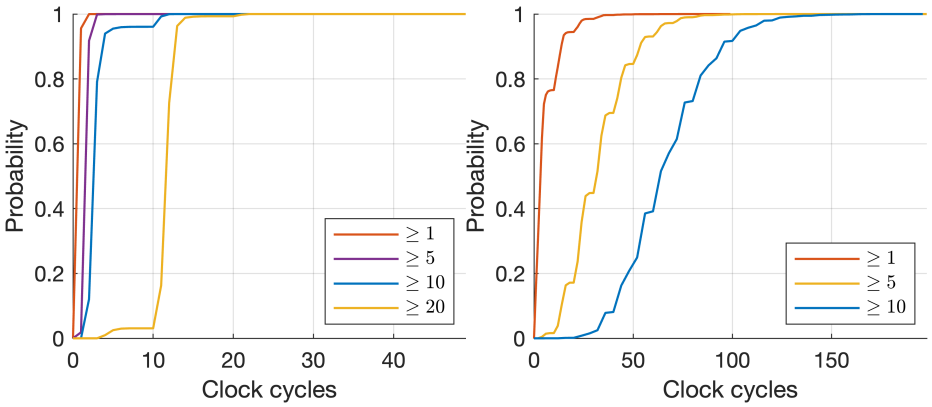
Additional results are shown in Figures 8, 9, and 10. These results are all generated using the statistical model checker `modes` in MODEST, and use the same configuration as the results in Section 7 (buffer size of four, 3/10 flit generation pattern, round robin scheduling, XY routing).

Similar to the analysis done in Section 7, we can see that the probability of reaching K PSN events grows with NoC size, as expected. Additionally, for the 3×3 and 4×4 the step pattern corresponding to the 3/10 flit generation pattern is clearly visible. For 8×8 results the step pattern is not pronounced, as it's more likely that the destination for a flit will be further away, thus keeping flits in the network for longer. Additionally, in an 8×8 NoC we observe that within three clock cycles we are likely to see 20 or more resistive noise events. This speaks to the large number of central routers in an 8×8 NoC that each have four neighbors, and as such are likely to have higher average traffic than corner or edge routers.



(a) Resistive Noise (b) Inductive Noise

Fig. 8. PSN CDF for 3×3 Modular Model



(a) Resistive Noise (b) Inductive Noise

Fig. 9. PSN CDF for 4×4 Modular Model

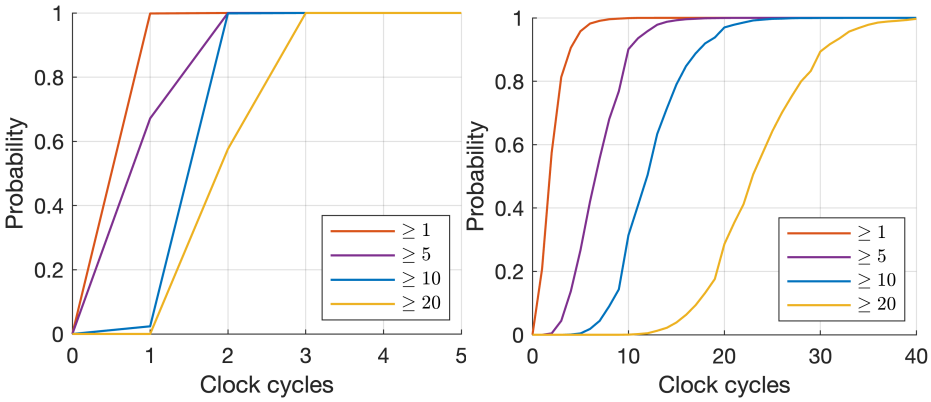
A.2 Datatypes

MODEST allows for the creation of high-level datatypes to represent state. We create several custom datatypes to define the structure and state of the NoC.

Router Datatype The router datatype defined in Code Segment 8 holds the state for a single router. This includes each buffer and channel, as well as data related to the arbitration algorithm (Section 4), and variables for tracking the number of PSN events (`activity`). In MODEST, the syntax `int(A..B)` defines an integer type in $[A, B]$.

Table 2. Description of router member variables

Member Variable	Description
`channels`	The buffer and channel instances.
`ids`	The IDs of neighboring routers.
`priority_list`	A priority queue of serviced routers
`priority_list_temp`	A temp queue used during calculation
`serviced_index`	
`unserviced_index`	
`total_unserviced`	
`thisActivity`	Activity counter for current clock cycle
`lastActivity`	Activity counter for last clock cycle
`used`	



(a) Resistive Noise (b) Inductive Noise

Fig. 10. PSN CDF for 8×8 Modular Model

```
datatype router = {
    channel[] channels,
    int(-1..NOC_MAX_ID)[] ids,
    int(0..4)[] priority_list,
    int(0..4)[] priority_list_temp,
    int(0..4) serviced_index,
    int(0..4) unserviced_index,
    int(0..5) total_unserviced,
    int thisActivity,
    int lastActivity,
    bool[] used
};
```

Code Segment 8. `router` Datatype

Channel Datatype The channel datatype defined in Code Segment 9 holds the information needed for correctly modeling a buffer in MODEST. The `option` modifier in MODEST declares a type that either holds a value or is empty.

Buffer Datatype The buffer datatype shown in Code Segment 10 is modeled as linked list, where there is some data stored with an optional tail pointing to the next element in the list. In this work, the data we store in the buffer is the same as the data in a packet in the network. Our packets only carry the ID of the destination router. The destination ID is valid from $[0, n^2 - 1]$ (Section 4) and we model that using a bounded int type in MODEST.

A.3 NoC State

The NoC state is stored as an array of router instances, where the index of each router instance corresponds to the ID of router. This state is stored as a global

```
datatype channel = {
    buffer option buffer,
    bool serviced,
    bool isEmpty,
    bool isFull
};
```

Code Segment 9. `channel` Datatype

```
datatype buffer = {
    int(0..NOC_MAX_ID) hd,
    buffer option tl
};
```

Code Segment 10. `buffer` Datatype

variable because each process needs to be able to access this state. The syntax for the `noc` router is shown in Code Segment 11.

```
router[] noc = [ /*... */ ]
```

Code Segment 11. NoC Data Structure

Global arrays in MODEST do not need to initialized at the start of model, however some of the CTL properties we specify in Section 5 must hold for all states, including the initial state. If the `noc` array is not default initialized, then the CTL properties fail on the initial state. Because of this, we explicitly initialize the `noc` array in the initial state as shown in Code Segment 12.

Other variables that track the state of the NoC are `resistiveNoise` and `resistiveNoise` counters. These counters track the total number of PSN events that happen in the system. Additionally, there is a integer `clk` that tracks the number of elapsed clock cycles in the system. This counter is necessary for measuring cycle-wise PSN, but must be removed during CTL model checking as we aim to check the system for unbounded clock cycles, as there are finite states but infinite clock cycles.

A.4 Processes

As described in Section 4, the NoC model is composed of modular processes that are composed together. Each process is modularized by taking an ID as a parameter. In each modular process, this ID is then used to read and update the router state in the `noc` array. For example, in a 2×2 NoC four router processes are composed in parallel, with each given a unique ID as shown in Code Segment 2. In each router process this unique ID is used to read and update the state of the router located at `noc[id]`.

Router Process The router process (Code Segment 13) is the *top-level* process in our hierarchical design. This process is made up of several sub-processes,

```
router[] noc = [
router {
    channels: [
        channel {isEmpty: true, isFull: false},
        channel {isEmpty: true, isFull: false},
        channel {isEmpty: true, isFull: false},
        channel {isEmpty: true, isFull: false},
        channel {isEmpty: true, isFull: false}],
    ids: [NO_CONNECT, NO_CONNECT, 1, 2],
    priority_list: [NORTH, EAST, SOUTH, WEST, LOCAL],
    priority_list_temp: [0, 0, 0, 0, 0],
    serviced_index: 0,
    unserviced_index: 0,
    total_unserviced: 0,
    thisActivity: 0,
    lastActivity: 0,
    used: [false, false, false, false, false]
},
/* ... continued for each router ... */ ]
```

Code Segment 12. Default Initialization

each of which execute a step in the router model, composed together in serial. Functionality over each clock cycle is done by a recursive call to the router process after calling the `nextClockCycle` action. Some sub-processes are described in further detail below, while others are commented in the code available in [16].

```
process Router(int id) {
    GenerateFlits(id);
    PrepRouter(id);
    AdvanceRouter(id);
    UpdatePiority(id);
    UpdateGlobalNoiseTracking(id);
    nextClockCycle;
    Router(id)
}
```

Code Segment 13. `Router` Process

Generate Flits Process As described in Section 4.5, the flit generation process can be adjusted to model unique flit generation algorithms. This process takes an ID, and can optionally add a new element to the local buffer of r_{ID}, provided that the local buffer of r_{ID} is not full.

Code Segment 15 shows the implementation of the 3/10 flit generation pattern, where a new flit is generated the first three out of every ten clock cycles. The basic outline of the process follows the template given in Code Segment 14, with one branch adding a new flit and the other branch doing nothing. `id` is

used to access a specific router instance in the `noc` array. In each branch the `generateFlits` synchronizing action is used to ensure that all state updates made by the `GenerateFlits` processes are atomic across all routers. This synchronizing action is required to minimize the state space and prevent write-before-read conflicts, as described in Section 4.3.

```
process GenerateFlits(int id) {
    if (/* local buffer is not full */) {
        /* conditionally add a new flit to the local buffer of r_id */
    } else {
        tau /* do nothing */
    }
}
```

Code Segment 14. `GenerateFlits` Process Template

In the 3/10 implementation shown in Code Segment 15, we ensure Property 3 from Section 5 by providing a uniformly random destination id that is shifted at our current id so as to not generate a flit for the originating router, and to maintain a uniformly random destination address.

```
action generateFlits;
process GenerateFlits(int id) {
 int(0..NOC_MAX_ID) destination;
 if (!isBufferFull(noc[id].channels[LOCAL].buffer)
     && clk % 10 < 3) {
  generateFlits {=
   0: destination = DiscreteUniform(0, NOC_MAX_ID - 1),
   1: noc[id].channels[LOCAL].buffer =
   enqueue(destination >= id ?
          destination + 1 :
          destination, noc[id].channels[LOCAL].buffer)
 =}}
 else { generateFlits }
}
```

Code Segment 15. `GenerateFlits` Process

Another possible implementation for the flit generation process is shown in Code Segment 16. This implemenation is specifically designed for a 2×2 NoC and introduces two new pieces of global state: the `burst_times` and `sleep_times` arrays. These arrays are indexed by the ID of a router, and contain a clock cycle count. Both counts start at zero. On the first clock cycle, both `burst_times[id]` and `sleep_times[id]` are initialized to a discrete random value in the bounds set by `[BURST_MIN:BURST_MAX]` and `[SLEEP_MIN:SLEEP_MAX]`. These values are how many flits will be injected and how many clock cycles no new flits will

be injected, respectively. This example uses the same uniform distribution of destination as the example in Code Segment 15.

This process can simulate a router in a NoC where there are "bursty" periods of activity where 10-100 flits need to be sent, followed by "slow periods" where nothing is sent. As an example, if in clock cycle 1, `burst_times[0]` is set to 20, and `sleep_times[0]` is set to 250, then over the course of the next 270 cycles r_0 will have 20 new flits injected, followed by a period where no new flits are injected for 250 cycles – assuming that the local buffer never gets full, in which case this whole period would be longer than 250 cycles.

Results from this "bursty" flit injection pattern are shown in Figure 11.

```
int[] burst_times = [0, 0, 0, 0];
int[] sleep_times = [0, 0, 0, 0];
const int BURST_MIN = 10;
const int BURST_MAX = 100;
const int SLEEP_MIN = 200;
const int SLEEP_MAX = 400;
process GenerateFlits(int id) {
 int(0..NOC_MAX_ID) destination;
 if (isBufferFull(noc[id].channels[LOCAL].buffer)) {
  generateFlits
 } else {
  // This custom process sends a flit in a bursty pattern
  if (burst_times[id] != 0) {
   generateFlits {=
    0: destination = DiscreteUniform(0, NOC_MAX_ID - 1),
    1: noc[id].channels[LOCAL].buffer =
       enqueue(destination >= id ?
         destination + 1 :
         destination, noc[id].channels[LOCAL].buffer),
    2: burst_times[id] = burst_times[id] - 1
   =}
  } else if (sleep_times[id] != 0) {
   generateFlits {=
    sleep_times[id] = sleep_times[id] - 1
   =}
  } else {
   generateFlits {=
    burst_times[id] = DiscreteUniform(BURST_MIN, BURST_MAX),
    sleep_times[id] = DiscreteUniform(SLEEP_MIN, SLEEP_MAX)
   =}
}}}
```

Code Segment 16. `GenerateFlits` Process

Prep Router Process The channel datatype in Appendix Section A.2 has two Boolean flags that indicate if that channel is empty or if the channel is full. These

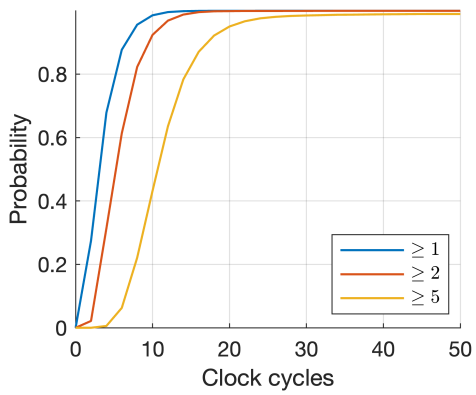


Fig. 11. Resistive PSN CDF for 2×2 Modular Model with Code Segment 16 Flit Injection Pattern

are set in the `PrepRouter` process (Code Segment 17) before flits are pushed to or popped from a buffer so that write-before-read conflicts do not occur.

```
action prepRouter;
process PrepRouter(int id) {
 // Prep all the channels
 prepRouter {=
  // North Channel (0)
  noc[id].channels[NORTH].isEmpty
    = len(noc[id].channels[NORTH].buffer) == 0,
  noc[id].channels[NORTH].isFull
    = isBufferFull(noc[id].channels[NORTH].buffer),
  /* channels 1-4 */
 =}
}
```

Code Segment 17. `PrepRouter` Process

Advance Router Process This process implements the routing algorithm described in Section 4.6. Describing a template for this process is challenging, as it's very flexible in what routing algorithms can be implemented. The key steps that must be implemented in this process are (1) flits that have reached their destination must be removed and (2) flits destined for other routers must be moved toward their destination (as long as there is a channel they can move on).

This work implements an X-Y routing algorithm, as described in Section 3.2. Additionally, this work uses a round-robin approach to arbiter traffic (Section 3.3). The `priority_list` in the router datatype (Appendix Section A.2) keeps

track of which buffers need to be serviced next, in order to ensure that all buffers are eventually serviced.

```
process AdvanceRouter(int id) {
    AdvanceChannel(id, noc[id].priority_list[0]);
    AdvanceChannel(id, noc[id].priority_list[1]);
    AdvanceChannel(id, noc[id].priority_list[2]);
    AdvanceChannel(id, noc[id].priority_list[3]);
    AdvanceChannel(id, noc[id].priority_list[4])
}
```

Code Segment 18. `AdvanceRouter` Process

Code Segment 18 shows the the implementation of `AdvanceRouter`. The advnace router process calls another sub-process `AdvanceChannel` for each element in the priority list. The updating of the priority list occurs the `UpdatePriority` (Appendix Section A.4). The specific MODEST code of the `AdvanceChannel` process are not given here, but detailed comments are available on [16] and algorithmic psuedocode is given in Algorithm 1 and 2.

Algorithm 1 `AdvanceChannel` Psuedocode

Require: Current router ID id
Require: Current buffer from priority list b
Ensure: If possible, the front flit of buffer b is either (1) removed if it's reached its destination, or (2) moved toward it's destination through a channel.

```
1: procedure AdvanceChannel(id, b)
2:     if isNotConnected(b) ∨ isEmpty(b) then
3:         return
4:     else if reachedDestination(peekFront(b), id) then
5:         f ← peekFront(b)          ▷ f would be sent to the connecting hardware.
6:         b ← dequeueFront(b)                          ▷ f gets dropped from b
7:         return
8:     else                                  ▷ Flit is destined for another router
9:         AdvanceFlits(id, b)
10:    end if
11: end procedure
```

Update Priority Process This process updates the priority list member variable of the `router` datatype and is used to arbitrate between conflicts between different buffers in a router. For example, if r_i^{South} and r_i^{North} each have a packet that is ready to send across r_i^{West} on the same clock cycle, the priority list will be used by the `AdvanceRouter` (Appendix A.4) to determine which packet is sent first.

A full discussion of the round robin semantics is given in [7].

Algorithm 2 `AdvanceFlits` Psuedocode with X-Y Routing

Require: Current router ID id
Require: Current buffer from priority list b
Require: The front flit of b ($peekFront(b)$) must not be destined for id
Ensure: If possible, a flit is moved b into a neighboring routers input buffer according to the X-Y routing algorithm.

```
 1: procedure AdvanceFlits(id, b)
 2:     Δx ← columnOf(id) − columnOf(peekFront(b).id)
 3:     Δy ← rowOf(id) − rowOf(peekFront(b).id)
 4:     if Δx = 0 then
 5:         if Δy > 0 then
 6:             Send flit north
 7:         else
 8:             Send flit south
 9:         end if
10:     else if Δx > 0 then
11:         Send flit east
12:     else
13:         Send flit west
14:     end if
15: end procedure
```

Clock Process The `clock` process is shown in Section 4.4, but the implementation is provided here as well in Code Segment 19.

```
process Clock() {
    nextClockCycle{= clk = (clk + 1) =};
    Clock()
}
```

Code Segment 19. `Clock` Process

Update Noise Process Code Segment 20 shows how the counters *resistiveNoise* and *inductiveNoise* are updated. If the change in activity is above the threshold, then *inductiveNoise* is incremented and another inductive PSN event is considered to have happened. If the activity in the current cycle is above the threshold, then a resistive noise event is considered to have happened and *resistiveNoise* is incremented. Finally, the activity counters are updated to have the current activity now stored as the previous activity.

A.5 Generating Unique Model With Python

Due to some of the complexity of managing a modular NoC model in Modest, we created a Python script that allows for repeatable generation of unique NoC models. This script is not required to create new NoC models, and models can

```
process UpdateGlobalNoiseTracking(int id) {
{=
   // Update inductive noise
   0: inductiveNoise +=
    abs(noc[id].lastActivity - noc[id].thisActivity) >= ACTIVITY_THRESH
    ? 1 : 0,
   // Update resistive noise
   0: resistiveNoise += noc[id].thisActivity >= ACTIVITY_THRESH
    ? 1 : 0,
   // Update trackers for next round
   1: noc[id].lastActivity = noc[id].thisActivity,
   2: noc[id].thisActivity = 0
=}
}
```

Code Segment 20. `UpdateNoise` Process

be updated by hand. However, it's simpler to automate this process to some extent, which is why we have provided the script. Code Segment 21 shows an example of generating a 2×2 NoC that can be used to characterize resistive noise with the MODEST TOOLSET.

```
from noc import *
_2x2 = Noc(2, buffer_size = 4,
             activity_thresh = 3, ...)
with open("2x2.modest", "w") as f:
  f.write(_2x2.print(PropertyType.RESISTIVE))
```

Code Segment 21. NoC Python Library

The `Noc` class is available in [16].

A.6 Adding a Unique Flit Generation Algorithm to the Model

Additionally, functionality is in place for unique flit generation algorithms to be added to models generated using the NoC class as shown in Code Segment 22. The unique flit generation algorithm must meet the requirements detailed in Section 4.5 and Appendix A.4, and should be encoded in plain text as a string in Python. Then to generate the model using the custom flit generation process, simply pass the string in to the Noc class `print` method to substitute the custom process in.

```
from noc import *
custom_generate_flits: str = """
int[] burst_times = [0, 0, 0, 0];
int[] sleep_times = [0, 0, 0, 0];
const int BURST_MIN = 10;
const int BURST_MAX = 100;
const int SLEEP_MIN = 200;
const int SLEEP_MAX = 400;
process GenerateFlits(int id) {
process GenerateFlits(int id) {
 int(0..NOC_MAX_ID) destination;
 if (isBufferFull(noc[id].channels[LOCAL].buffer)) {
  generateFlits
 } else {
  // This custom process sends a flit in a bursty pattern
  if (burst_times[id] != 0) {
   generateFlits {=
    0: destination = DiscreteUniform(0, NOC_MAX_ID - 1),
    1: noc[id].channels[LOCAL].buffer =
       enqueue(destination >= id ?
         destination + 1 :
         destination, noc[id].channels[LOCAL].buffer),
    2: burst_times[id] = burst_times[id] - 1
   =}
  } else if (sleep_times[id] != 0) {
   generateFlits {=
    sleep_times[id] = sleep_times[id] - 1
   =}
  } else {
   generateFlits {=
    burst_times[id] = DiscreteUniform(BURST_MIN, BURST_MAX),
    sleep_times[id] = DiscreteUniform(SLEEP_MIN, SLEEP_MAX)
   =}
  }
 }
}
"""
_2x2 = Noc(2);
_2x2_as_str = _2x2.print(PropertyType.RESISTIVE,
                      generate_flits=custom_generate_flits)
```

Code Segment 22. Using the Python Library to Generate Custom Flits