PROBABILISTIC VERIFICATION FOR MODULAR NETWORK-ON-CHIP SYSTEMS

by

Jonah W. Boe

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Zhen Zhang, Ph.D. Major Professor	Arnd Hartmanns, Ph.D. Committee Member
Sanghamitra Roy, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2023

ABSTRACT

Probabilistic Verification for Modular Network-on-Chip Systems

by

Jonah W. Boe, Master of Science

Utah State University, 2023

Major Professor: Zhen Zhang, Ph.D.
Department: Electrical and Computer Engineering

Modern network-on-chip (NoC) systems require better modeling tools in order to further understand issues caused by power supply noise (PSN), and to aid in mitigating the issues that heavy system traffic can have. Dramatic variations in network traffic leads to fluctuations in the power delivery across the entire chip, which can ultimately lead to errors in data transfers between the individual routers. Further complicating the problem, NoC applications vary widely in size and traffic patterns. Thus, constructing a formal NoC modeling system whose properties can be verified with minimal effort becomes advantageous for streamlining design and verification procedures. This thesis provides a novel approach to modularizing the design of probabilistic NoC models by creating a universal router as the fundamental component, capable of being instantiated into larger networks with minimal effort. Additionally, formal properties are developed to ensure the functional behavior of the modular design, and a structured approach is presented for ensuring that NoC models are indeed representative of their physical counterparts. The modular model is then scaled and used to analyse PSN properties in a larger topology, not previously verified probabilistically. The Modest Toolset is used for stochastic modeling and verification.

(78 pages)

PUBLIC ABSTRACT

Probabilistic Verification for Modular Network-on-Chip Systems

Jonah W. Boe

Modeling physical systems with formal analysis tools can help in the design of more fault-proof systems, by helping to determine if unpredictable or unwanted behavior may occur. Probabilistic verification further advances such processes, by providing quantitative information about the system. More complex systems can especially benefit from formal modeling and verification, as testing the physical system in every possible condition manually, can be extremely complex, and often impossible.

There is a growing interest in the application of Network-on-Chip (NoC) systems. NoCs can help simplify communication between the subsystems of many technologies, including the ever more complex multicore processors being produced. These NoCs come with their own problems, and under high network activity, can cause power fluctuations on the chip's power supply. These fluctuations can cause data corruption and loss, resulting in reduced performance, and even unpredictable behavior.

This work presents a novel approach to creating a modular probabilistic model of an NoC, which can be scaled to meet the needs of a variety of implementations. Additionally, it presents a structured approach for ensuring that NoC models are indeed representative of their physical counterparts.

To my wife and kids. For all of their patience and love.

ACKNOWLEDGMENTS

During the course of my education, I have been influenced and inspired by many friends, family, and educators. The work presented here would not have been possible without their support.

Primarily, I would like to thank my advisor Dr. Zhen Zhang for providing me with this great opportunity, and for all of his many contributions along the way. His expertise and guidance has been invaluable to me throughout this program.

I would like to thank the faculty and staff of the ECE department here at USU, for their guidance and support. I especially want to thank Tricia Brandenburg, Diane Buist, and Kathy Phippen, for their administrative efforts on my behalf.

I would like to thank my committee, for their time and efforts on my behalf. Especially Arnd Hartmanns, for his professional advice on working with the modeling tools used in this work.

I would like to thank my parents (Edward and Lillith Boe), for their love and support. Especially for their encouragement, in persuit of my passions in engineering.

Finally, I would like to thank my wife Abby, for encouraging me and supporting me throughout my higher education. She has made the best of even tough circumstances, and brought joy to every moment.

Jonah W. Boe

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

CMOS	Complementary Metal-Oxide-Semiconductor
DTMC	discrete-time Markov chain
FML	formal modeling language
LTL	linear temporal logic
MPSoC	multiprocessor system on chip
NoC	network on chip
PCTL	probabilistic computation tree logic
PSN	power supply noise
SoC	system on chip

NOTATION

Routing

r_i	router with ID of i
r_i^b	buffer b (Local, North, East, South, West) of r_i
$x \twoheadrightarrow r_i^L$	new flit with destination x is generated by the Local buffer of r_i
$r_i^s \xrightarrow{x} r_j^d$	flit with destination x is routed from the source buffer r_i^s to the destination buffer r_j^d

CHAPTER 1
INTRODUCTION

As modern systems grow in complexity, there has been an increasing need for highly efficient on-chip communications, and the days of all communications on a chip occurring through a single data bus are quickly becoming antiquated. Network-on-chip (NoC) architectures are quite advantageous in this regard. Similar to routers in traditional computer networks, they allow for multiple nodes within a single chip to communicate simultaneously. Such a solution can vastly improve the overall communication throughput of a system, compared to traditional bus-based methods. Traditionally, a singular data bus (set of connections comprised of chip-select, clock, and data lines) would carry information between sub-systems of a chip. If one sub-system needed to communicate with another, it would check to make sure that the bus was not in use by others before transmitting. Such systems become overwhelmed when many nodes demand the bus at once. Implementing an NoC allows for a more intricate system of connections between chip sub-systems. These nodes are capable of communicating with direct neighbors, without interfering in other communications occurring within the network.

Because of the advantages NoCs provide, a great deal of work is currently going into perfecting their performance, especially in complex architectures, such as system-on-chip (SoC) designs [1, 2]. These advanced chips take advantage of modern silicon production capabilities. This in turn allows for single chips to contain many more sub-systems, from standard processing units to specialized hardware accelerators [3].

Though they are becoming highly utilized throughout many industries, NoCs still face new challenges not prevalent in their bus-type counterparts. For NoCs to truly become a viable solution, especially in safety-critical systems, there needs to be guarantees about the specific properties that govern such systems, as any faults in their design can lead to catastrophic effects [4, 5]. One of the major properties inherent in current NoC systems is that of power supply noise (PSN). PSN occurs when there are sudden shifts in network packet traffic activity across individual nodes. Such sudden changes in traffic patterns directly correspond to fluctuations in power drawn by those nodes. Larger fluctuations in power draw can result in data corruption or loss. By integrating formal modeling and verification approaches early within the NoC design flow, problems such as PSN may be mitigated, and guarantees can be made as to their functional and quantitative correctness. Formal modeling gives great insight into the intricate details of NoC design, even helping to eliminate bugs and errors early on in the NoC design process. This is important, as patterns of PSN are not consistent between NoCs, and traffic patterns can vary widely from application to application. Thus, it becomes advantageous to be able to have quantitative guarantees about the likelihood of PSN across a variety of systems and applications.

While formal modeling of NoCs continues to be studied, limited work has been done to model them probabilistically [6]. Probabilistic checking is not only used to verify invariants, but more especially, to gain insight into the quantitative likelihood of particular events. This work proposes that such modeling is advantageous to better understanding issues associated with PSN, as non-probabilistic model checking does not reveal quantitative information about the properties being checked. This information can assist in determining the criticality of unwanted events. Additionally, existing probabilistic NoC modeling solutions are lacking in modularity, with the entire NoC systems being modeled in a single monolithic module [7,8].

The NoC router model presented in this work is synchronous in nature. That is, all routers in the model execute in parallel, synchronizing flit generation, buffer updates, and arbitration. At the time of this writing, there does not appear to be a synchronous quantitative formal modeling language that can directly meet the need of this work – modeling a synchronous NoC and verifying probabilistic properties. For this reason, a number of additional variables (e.g. a clock and flags), were coded into the NoC router modules, in order to achieve the correct behavior. The Modest formal modeling language is used in this work, as at the time of this writing, it appears to be the only formal modeling language with support for arrays and lists. These datatypes are critical to NoC design, as they make up a large portion of the system architecture.

1.1 Contributions

This work provides a modular formal modeling approach for NoC designs. Such a model can be scaled to contain more routers with minimal effort for easy analysis of larger scale implementations. Additionally, it provides functionality verification at a local level to ensure that the subcomponents of the modular NoC are behaving as intended. At a higher level, procedures are outlined for verifying system correctness on an application basis.

The major contributions of this work include:

- The derivation of a modular probabilistic model for measuring the PSN of a NoC, by creating a NoC model that can be reconfigured to meet a variety of needs with minimal effort.

- A set of formal properties for ensuring the functionality of each module of the modular NoC. This results in proven correct modular components that can be instantiated. To accomplish this, the Modest Toolset is utilized for the statistical model checking of LTL properties.

- A novel procedure for verifying the modular NoC on an application specific basis. This is accomplished by utilizing data collected from other working models of the system, or statistics from previous implementations or prototypes. Unlike discrete modeling, verifying the equivalency of two probabilistic models is beyond property checking. While such tasks may help in identifying differences and bugs, this work proposes a more structured approach which utilizes the probabilistic results for identifying discrepancies.

- Formal verification of a 3×3 NoC. Using this new model, analysis is performed for identifying patterns of PSN within a 3×3 NoC model using the statistical verification tools available in the Modest Toolset. The results of this verification reveals that larger scale NoCs are inherently more prone to PSN than smaller implementations, assuming uniform packet injection patterns.

1.2 Thesis Outline

The remaining work is as follows. Chapter 2 investigates related work in this field. Chapter 3 gives the necessary background information including the NoC architecture and routing procedures, and a detailed description of statistical model checking techniques and procedures. Chapter 4 outlines the motivation which prompted this work. Chapter 5 describes the design decisions made in developing a modular NoC. Chapter 6 gives a detailed walk through on scaling the modular NoC. Chapter 7 elaborates on the specific properties used to verify the functional correctness of the modular NoC. Chapter 8 describes how existing data can be used for ensuring that a modular NoC implementation is representative of the system it aims to emulate. Chapter 9 analyses patterns of PSN within a 3×3 topology. Chapter 10 concludes this thesis and presents possible future research opportunities.

CHAPTER 2
REVIEW OF LITERATURE

Modern formal verification techniques are quite capable of providing provable guarantees to real-world applications, and can lend valuable insights into the properties of the systems under verification. Modern NoC technologies currently benefit from powerful formal verification tools, resulting in more robust fault-resilient systems. Such verification techniques have been used especially in the high-level behavioral design stage of these systems. Formal verification can continue to be of great value in helping to further mature this technology by providing quantitative information about specific NoC implementations within the application stage of such systems.

2.1 Formal Verification of NoCs

Much of the existing formal verification work on NoC designs has used non-probabilistic verification techniques. Such techniques are useful for verifying the correctness and functionality of the NoC systems. Functional correctness properties for NoCs include verifying correct routing of flits [9, 10] and securing vulnerabilities [11].

While limited, some research has been done to verify NoC systems using probabilistic techniques. The work of [12] verifies their overall performance in heterogeneous CPU-GPU chip architectures. It focuses on using optimized routing algorithms to improve system performance and to lower chip power consumption. An algorithm is proposed based on the Strength Pareto Evolutionary Algorithm2 [13], boasting a 17% improvement on performance, while decreasing power consumption by at least 2.3 times that of its predecessor.

Two particular problems facing NoC development are deadlock and livelock. Deadlock occurs when a buffer is starved as the result of a cyclical dependency within the NoC routing protocol. Livelock problem occurs when a routing algorithm causes a flit to be cycled throughout the system without ever reaching its destination. The work done in [14] presents a routing algorithm which builds on the Glass/Ni fault-tolerant routing algorithm [15]. In addition to maintaining the livelock freedom of the Glass/Ni algorithm, it improves the algorithm by adding deadlock freedom. By verifying the livelock-freedom functionality of NoC routing algorithms, more can be understood about what traffic patterns are better suited for certain applications; improvements can then be made; and better routing practices can be developed [9]. This latter work presents a proven correct packet routing protocol, which patches a scalability issue found in the protocol presented in [14]. This issue resulted in deadlock for topologies larger than 2×2.

As the primary hub for communications in modern systems, NoCs are a prime target for attacks. Such attacks include prevention of data to travel between components, data leaks, and corruption of system behavior. The formal verification of NoCs can prove the correctness of security features, or even reveal further vulnerabilities not yet perceived. Such vulnerabilities are critical to resolve, especially in systems such as multiprocessor system on chips (MPSoCs), where NoCs are being used to communicate potentially sensitive information between the cores of multiprocessors [11].

As a result of the work being done to formally verify the many properties of NoCs, not only has reliability of these systems improved, but also their energy footprint [12]. This is especially critical in high density applications, like MPSoCs. In such applications, large amounts of data are being communicated between processors at very high rates. The switching of logic gates causes peaks in current draw, resulting in increased thermal output by the NoC. Lowering the power consumption of these systems can greatly decrease their thermal output and improve overall system efficiency [16].

2.2 Probabilistic Model Checking

Unlike their deterministic counterparts, probabilistic modeling tools can be used to verify both *if* a system can reach a certain state of interest and the *probability* of such an event occurring. For example, one might be interested in ensuring that a system design can reach some particular state. Such an assertion might hold true using non-probabilistic model checking tools. However, modeling the same system using probabilistic tools can reveal that, for example, the probability of reaching the desired state was actually below some arbitrarily low percentage, which might be unacceptable. With this information, the system could be redesigned until the more robust probabilistic property is satisfied. This is the advantage of probabilistic model checking. It can provide more detail about the quantitative metrics of a system than its deterministic counterparts [17].

A fair amount of research has been going into developing probabilistic verification tools, with the earliest reference appearing in 1983 [18]. Modern research has produced tools such as Modest [19], PRISM [20], Storm [21], IscasMC [22], and PAT [23], to name a few. The work of [20] elaborates on the powerful capabilities of PRISM, for example, as a tool for modeling more complex embedded systems. The Modest Toolset has similar modeling capabilities, and aims to integrate existing state-of-the-art tools into a single environment [19]. Most probabilistic model verification tools can be applied to a variety of fields, ranging from flight controllers [24] to molecular biology [25]. The versatility of probabilistic modeling tools has not only gained them attention in both industry and academia, but also provided an opportunity for such tools to be improved, as feedback is received from professionals of diverse backgrounds.

While probabilistic model checking has many advantages, it does come with some limitations. One of its main drawbacks is that it tends to require significant computational resources. This issue is referred to as the state space explosion problem, and a practical solution has yet to be developed. Model checking is the processes of enumerating every reachable state of a system, and the state space increases exponentially as the number of variables and processes grows. Of course, the usefulness of probabilistic model checking is greatly reduced when constrained to only very simple systems, so many potential solutions have been proposed. Many of these solutions exist especially for simplifying finite-state systems. These include symmetry reduction [26], symbolic model checking [27], and bisimulation minimization [28]. With regard to infinite-state systems, these approaches are not applicable unless the model is truncated. Truncation attempts to represent an infinite-state system using a finite approximation. This process is tedious when done manually, and attempts to automate it on certain models can also result in state space explosion [29,30].

2.3 Statistical Model Checking

Statistical model checking uses data collected from a relatively large set of random simulation runs in order to approximate a model's probabilistic properties [31]. As a result, the confidence level of these approximations is proportional to the number of runs executed, with more executions resulting in higher confidence. Even so, models which contain extremely rare events may take hundreds of thousands of runs to achieve accurate results. Even so, the certainty of results can never be guaranteed.

While statistical model checking does not provide the same level of certainty, as opposed to probabilistic model checking, it does provide some major advantages. First, because it does not explore the entire state space of the system, only the applicable results of each run need to be stored in memory. Second, it is much simpler to take advantage of parallel processing when implementing this approach, as each simulation of the system model runs independent of the others. Lastly, this approach is easily applicable to a broad range of tools, regardless of the underlying logical reasoning method being used [32]. These benefits mean that statistical model checking is not only much less resource intensive, but it can also be scaled for use in much more complex systems.

In order for the probability of rare events to be estimated, the event itself must be known, and expressed as a property of the system. Rare-event simulation then gives some bias to that event while simulating. This does skew the results however, and techniques must be used to normalize them [33]. Rare-event simulation cannot guarantee results, however tools like Modest do give the user the ability to set the confidence interval. A higher confidence interval inherently requires more time and resources.

The Modest Toolset is the primary tool used in this work, and while it has the capability of doing both probabilistic and statistical model checking [19,34], this work will primarily use the latter for its analysis. Additionally, the Modest Toolset is capable of rare-event simulation through automated importance splitting [35], though these tools will not be used in this work.

2.4 Probabilistic Verification of NoCs

As of this writing, there has been very little existing work on the formal verification of NoCs using probabilistic verification techniques. One major contribution is the model checking of reliability properties of a single central router of an NoC. The work of [8] examines the possible states of a central router and the probabilities of transitioning between those states under uniform traffic load. For example, given any two possible states a and b, what is the probability that a will directly proceed b? The work showed that the single router could be represented by a table of transitions, and that by using this transition table alone, information about the PSN of the individual router could then be determined. By implementing all state transition probabilities as a set of deterministic choices, a model was designed to check resistive and inductive noise properties. The work proved that probabilistic model checking is capable of providing valuable insight into the field of PSN in NoCs.

Additional work has been done to create an abstracted model of a 2×2 mesh network topology. The work of [7] added two major contributions. First, the state transition probability matrix from the previous work was used to develop a complete 2×2 NoC. Second, this model was abstracted using probabilistic choice abstraction, a novel concept which is achieved by eliminating unnecessary variables in order to combine otherwise identical state paths within the individual branches of execution. By doing this, the overall system state space was greatly reduced.

The probabilistic choice abstraction introduced in [7] does address scalability to some extent, although the work itself focuses on only the 2×2 NoC. Additionally, the models developed in these works are monolithic, and therefore, are not broken down into submodules. Because of this, their correctness was validated manually. While this approach was feasible for their smaller topologies, such an approach is not easily scalable. By designing the modular NoC using a non-monolithic approach, the work presented in this paper allows for the verification of the systems submodules to be automated. Additionally, the modular approach allows for a level of scalability never before achieved in a probabilistic NoC model.

CHAPTER 3
BACKGROUND

This chapter gives a description of background knowledge relevant to this thesis. Section 3.1 describes how flits are routed within the system, including routing conflicts and resolution. Sections 3.2 to 3.4 outline the formal modeling principles required in order to understand this work. Finally, Section 3.5 gives an overview of the Modest Toolset, including language syntax, and property checking.

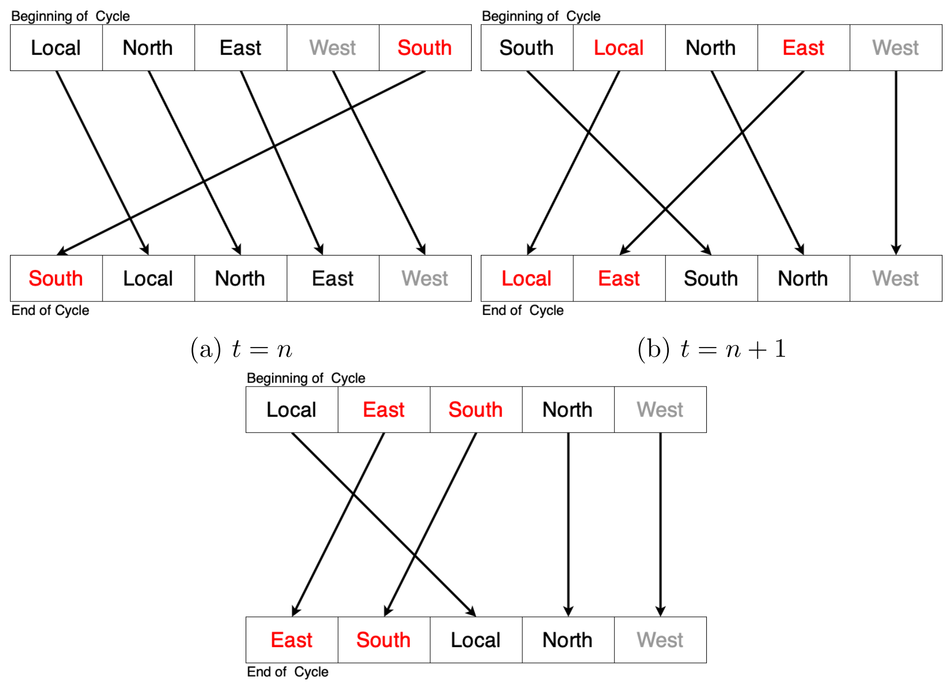
3.1 Flit Routing

In this work, router inputs contain buffers and are therefore referred to as inputs and buffers synonymously. Likewise, the outputs and channels will be used interchangeably.

In traditional networking, packets are used to communicate information between routers or transceivers. These packets have a standardized size, and are given a header, which includes the addresses of the both the sender and the recipient. This packet can then be routed through a link of other routers until it has reached its destination. There, the packet is unpacked from its header, decompressed if necessary, and put to use depending on application. Routing within an NoC has a similar purpose and implementation. Flits containing information are sent from one router to another in order to transport information between components of the overlying system. In order to accomplish this, the flit must at least include the destination router address, so that the routers in the system know where to send it.

Similar to traditional networking, the path a flit takes to get from the sender to the recipient depends on the protocol that has been implemented across the network. As mentioned, this work uses X-Y routing. This means that flits are first routed to the west or east, until they have reached the same column as the destination router. Only then are they routed north or south until they have reached their destination. Once a flit begins routing in the Y direction it cannot switch back to routing in the X direction. Additionally, flits cannot be routed away from their destination, nor past their destination for any reason. Meaning that a flit cannot be routed east *and* west within its lifetime. The same is true for routing north *and* south. For example, a flit being routed from r_3 to r_8 within the configuration shown in Fig. 6.1, will always take the X direction path $r_3^L \xrightarrow{8} r_4^W$, $r_4^W \xrightarrow{8} r_5^W$, before taking the Y direction path $r_5^W \xrightarrow{8} r_8^N$. When flits reach their destination router, they are then removed from the system by that router.

By its very nature, X-Y routing does not allow for cyclical routing behavior, eliminating the opportunity for livelock. There is, however, still the possibility that starvation can occur. This could happen, for example, if the east buffer of a router is competing for the west channel every cycle, and is allowed priority over the other routers repeatedly. Similar to the work done in [8], this model uses a Round-Robin pattern for the resolution of such conflicts, by assigning higher priority to the buffers that are not serviced in a given cycle. An example of a priority list update sequence is illustrated in Fig. 3.1. For this example, it is assumed that there are three consecutive cycles of flit injection for a single buffer. For reference, see r_3 in Fig. 6.1. The west buffer is grayed out to indicate that it was not assigned a neighbor, and buffers in red were not serviced in the given cycle. While other conflicts could be assumed, here it is assumed that the south buffer conflicts with the local buffer in cycle (a), as shown in Fig. 3.1. Because the south buffer is farther towards the end of the priority list, it is the one marked as unserviced for the next cycle. It is then assumed that in cycle (b) of Fig. 3.1, the south, local, and east buffers have all competed for the north output channel. In this scenario, the south has the highest priority and is therefore serviced, while the local and east buffers are advanced in the priority queue. In cycle (c) of Fig. 3.1, in addition to the last conflict, the south buffer now requires the north output channel again. It is reasonable that it should not take priority over the east buffer, as the east buffer has now been waiting for a longer duration. Notice that by marking the west buffer as serviced every cycle – though it is not connected to a neighbor – it has been pushed to the back of the priority list, and is consistently kept at lower priority than those buffers which are connected to neighbors. In this way, unassigned buffers never compete for priority. This holds true for routers with more than one unassigned neighbor as well, such as is the case for routers r_0, r_2, r_6, and r_8 of the same system.



(a) $t = n$

(b) $t = n + 1$

(c) $t = n + 2$

Fig. 3.1: Example of Round Robin Conflict Resolution

For the individual router to know which direction to send a flit next, information about the dimensions of the system must be available to every router. This is accomplished by keeping the maximum router ID (1 - the total number of routers) as a global variable, as all routers will need to be able to access it. Because the system must be $n \times n$, the router can use this information to determine not only the width and height of the system, but also its own position within that system.

3.2 Modeling Discrete-time Markov Chains

This section introduces concepts of modeling discrete-time Markov chains (DTMC)s. DTMCs can be defined as a tuple $(S,\ \bar{s},\ \mathbf{P},\ L)$, where S is a finite set of states describing some system, and $\bar{s}$ is the initial state. Additionally, $\mathbf{P}$ is the probability matrix describing all possible state transitions of the system, such that each row of $\mathbf{P}$ describes all of the transitions for a single state. Note that the sum of the transition probabilities of a single state must add to 1. That is, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ must hold for all elements of $\mathbf{P}$, where s' is the next state, and $\mathbf{P}(s, s')$ is the probability of transitioning to the next state. Lastly, L is a function which assigns to state s the appropriate atomic propositions of that state.

Fig. 3.2 shows a graphical representation of a DTMC for the game rock-paper-scissors. For this example, $\mathbf{P}$ is given by Equation 3.1. The state labels have been assigned: $L(s_0) = \varnothing$, $L(s_1) = \{rock\}$, $L(s_2) = \{paper\}$, and $L(s_3) = \{scissors\}$. There is a uniformly distributed probability that a player goes to any one of the three states (rock, paper, scissors) from the starting state S_0. The player then returns to the start for the next round. Assuming that there is interest in the probability of the path $S_0 \rightarrow S_1 \rightarrow S_0 \rightarrow S_2$, being taken by the player. This probability can be calculated as the product of all state transitions over the path, or $1/3 \cdot 1 \cdot 1/3 = 1/9$.

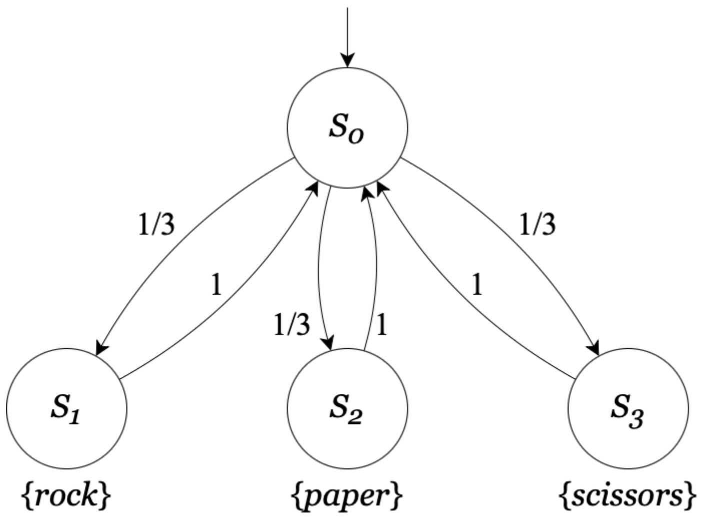


Fig. 3.2: DTMC Example

$$\mathbf{P} = \begin{pmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{3.1}$$

3.3 LTL Property Specifications

Linear temporal logic (LTL) formulas use temporal and logical operators, in conjunction with propositional variables, to describe not only the state reachability of systems, but more practically, temporal patterns such as sequencing, recurrence, and persistence. The basic temporal operators of LTL are shown in Table 3.1. Here a and b are assumed to represent known, unique state formulas. These formulas represent the evaluation of automatic propositions, and must have a boolean result. x represents another state formula, and does not have any defined relationship to a or b.

Operator	**Description**	**Example Trace**
a	a	$a \rightarrow x \rightarrow x \rightarrow x \rightarrow x \rightarrow x$
$\bigcirc a$	next is a	$x \rightarrow a \rightarrow x \rightarrow x \rightarrow x \rightarrow x$
$\Diamond a$	eventually a	$x \rightarrow x \rightarrow x \rightarrow x \rightarrow a \rightarrow x$
$\Box a$	always a	$\rightarrow a \rightarrow a \rightarrow a \rightarrow a \rightarrow a$
$b \cup a$	b until a	$b \rightarrow b \rightarrow b \rightarrow a \rightarrow x \rightarrow x$

Table 3.1: Temporal Operators

Some additional logic operators can be used to further describe the behavior of the system. A few of these are shown in Table 3.2.

3.4 PCTL Property Specifications

Properties of DTMCs can be specified using probabilistic computational tree logic (PCTL). PCTL properties build on LTL principles using some additional syntax. The syntax most important to this work is $\mathbf{P}_{\sim q}(\Diamond^{\leq K}\Psi)$. That is, the probability that Ψ is eventually true within K time steps satisfies the probability interval specified by $\sim q$. $\sim q$

Operator	**Description**
$\neg a$	true when a is false
$a \vee b$	true when a or b is true (both can be true)
$a \wedge b$	true when a and b are true
$a = b$	true when a is equal to b
$a < b$	true when a is less than b
$a > b$	true when a is greater than b
$a \leq b$	true when a is less than or equal to b
$a \geq b$	true when a is greater than or equal to b

Table 3.2: Logical Operators

is further defined by $\sim \in \{<, \leq, \geq, >\}$, and $q \in [0, 1]$. Additionally, $\mathbf{P}_{\sim q}$ can be written as $\mathbf{P}_{=?}$, indicating that the probability is the variable of interest. For a complete definition of PCTL syntax and semantics, see [36].

In the example shown in Property 3.2, the property will evaluate to true, if the probability that a path will eventually contain a state where a is true, is less than 25%.

$$\mathbf{P}_{<0.25}(\Diamond a) \tag{3.2}$$

In the example shown in Property 3.3, the property will evaluate to the probability, that all paths will never contain a state where a is equal to b.

$$\mathbf{P}_{=?}(\Box(\neg(a = b))) \tag{3.3}$$

There are occasions were the PCTL property might be of interest only within a certain number of time steps. This is important, as an individual run can technically execute indefinitely. The syntax used to describe PCTL specification in this work is $[\text{accumulate}(clk) \leq N]$. In the example shown in Equation 3.4, the property will evaluate the probability, that within 10 clock cycles, all paths eventually contain a state where a is true.

$$\mathbf{P}_{=?}(\Diamond^{[\text{accumulate}(clk) \leq 10]}\, a) \tag{3.4}$$

3.5 The Modest Toolset

The Modest Toolset provides both the Modest modeling language for formal modeling systems, and a suite of tools for model simulation and verification. This section describes the Modest language syntax and the analysis tools that are relevant to this work.

3.5.1 Modest Language and Syntax

The Modest language contains a handful of primitive datatypes, including *int*, *real*, and *bool*. Both *int* and *real* can be bounded upon initialization. Code Segment 3.1 shows an example of how such bounding is implemented.

```
int (0..5) x;
```

Code Segment 3.1: Bounding of Integers in Modest

Arrays are also supported. Their syntactic implementation is similar to that of arrays in C-type programming languages. An example implementation is shown in Code Segment 3.2.

```
int[] x = [0, 1, 2, 3];
```

Code Segment 3.2: Array Instantiation in Modest

Additionally, Modest includes a method for simplifying array Instantiation. An example is shown in Code Segment 3.3. This implementation creates an array x, of five objects of type j.

```
j[] x = array(i, 5, j{});
```

Code Segment 3.3: Array Method in Modest

In addition to the primitive datatypes, Modest allows for the implementation of custom datatypes. These datatypes are similar to *structs* in C-type programming languages, and can consist of both primitive datatypes, and other custom datatypes. Code Segment 3.4 demonstrates an example implementation.

```
datatype x = {
    int i,
    bool j
};
datatype y = {
    int id,
    y[] z
};
```

Code Segment 3.4: Custom Datatypes in Modest

It should be noted that the Modest language consists of two additional keywords: *none* and *option*. The keyword *none* can be assigned to a variable to indicate that it is not currently assigned. Notice that in Code Segment 3.1, x was created, but not initialized. This does not make x equal to *none*. The assignment to *none* must happen explicitly. In order to assign a variable as *none*, it has to have been created using the *option* keyword; indicating that the variable can be assigned the value of none. Only variables created with the *option* keyword can be used when assigned to *none*.

The Modest language includes two functional datatypes: functions and processes. Functions are used when a value needs to be returned, and can only be used to evaluate the parameters explicitly passed in. This means that neither variable creation, nor assignment is allowed within the scope of a function. Processes, on the other hand, are allowed to include assignment operators and local variables. Alternatively, they are not allowed to return a value. As a result of these rules, functions are generally used to create reusable logical and arithmetic operations. Generally, processes define the behavior of the system by manipulating variables.

Until now, the examples given have demonstrated variable assignment during either initialization of a global variable, or in the description of a custom datatype. It is important to note that assignments within processes have special rules. First, assignments are grouped into blocks. These blocks are indicated by special operators, as shown in Code Segment 3.5.

Second, these assignment blocks are what tell the Modest Toolset to store a new state. Additionally, although the assignments made in these blocks happen concurrently, some order can be explicitly given. An example of this is shown in Code Segment 3.6. In this example, the two lines labeled 0 execute concurrently, followed by the next two lines in sequentially in the order given.

```
{=
    i = 5,
    j = false
=}
```

Code Segment 3.5: Process Assignment Operator Block

```
{=
    0: a = 5,
    0: b = false,
    1: c = 2,
    2: d = 2
=}
```

Code Segment 3.6: Ordered Process Assignment Operator Block

Using available Modest keywords, lists can also be created. There is not a native datatype for lists in Modest, but they can easily be constructed as shown in Code Segment 3.7. As of this writing, the Modest language appears to be the only probabilistic modeling language with support for both arrays and lists. This makes Modest even more ideal for this work, as the priority lists and buffers require such datatypes. List manipulation and access are both accomplished through custom functions. Code Segment 3.8 shows how to add items, and Code Segment 3.9 shows how to access the front item in the list. The *tl*, or tail, is the main body of the list. The *hd*, or head, is the first item in the list. Additional functions can be created for checking the *hd*, removing the *hd*, checking if the list is empty, and checking the length of the list.

```
datatype list = {
    int hd,
    list option tl
};
```

Code Segment 3.7: Creating Lists in Modest

```
function list option enqueue(int i, list option lst) =
    some(list {
            hd: i,
            tl: lst
    });

process addItem(int i){
    list lst;
    {=
        lst = lst.enqueue(i)
    =}
}
```

Code Segment 3.8: Adding Item to Lists in Modest

```
function int peekFront(buffer option ls) =
    if ls == none then -1
    else if ls!.tl == none then ls!.hd
    else peekFront(ls!.tl);

process getFront(list lst){
    int front;
    {=
        front = lst.peekFront()
    =}
}
```

Code Segment 3.9: Getting Front Item of Lists in Modest

In this work, each router in the NoC is given its own process, and all are run in parallel. The parallel composition of all router processes is achieved in Modest through the *par* operator. Code Segment 3.10, gives an example. Note that the *Player* and *Ball* must be processes. In this example, the *Player* process is instantiated twice (once for each player), and the *Ball* is instantiated only once. Also note that all three processes are run in parallel.

```
par{
    :: Player()
    :: Player()
    :: Ball()
}
```

Code Segment 3.10: Ordered Process Assignment Operator Block

Actions are also available in Modest. The one built-in action used in this work is *tau*. It is a silent action, meaning that it simply serves to be a placeholder where nothing is desired to happen. In Modest all *if* statements must have a corresponding *else* statement, though the may be separated by many *else if* statements. Additionally, no conditional block can be left empty. Code Segment 3.11 shows a proper use case for the *tau* action.

The two non-silent actions provided by Modest are *break* and *error*. The only construct for looping is *do*. The *do* construct will execute until either a *stop* or *abort* behavior is reached, or a *break* action is performed. Code Segment 3.12 shows an example where the loop exits after five cycles in the loop. The *error* action can be used to indicate an error has been reached, and is actualy called by the *abort* behavior.

```
if(a > b){
    {=
        c = 5
    =}
}
else{
    tau
}
```

Code Segment 3.11: Ordered Process Assignment Operator Block

```
{= i = 0 =};
do{
  if(i < 4){
        {= i++ =}
    }
    else{
        break
    }
};
```

Code Segment 3.12: Example of a *do* Loop in Modest

Custom actions are also available, and in this work, they are used to synchronize any common parallel processes. By default, actions use multi-way synchronization, meaning that all parallel processes containing the same action are halted until all processes are ready to execute that action. Once all processes are ready, they all execute the action simultaneously in a single step. Not all process in parallel composition have to contain the same synchronizing actions, and multiple actions can exist in any number of processes. Custom actions must be declared at the global scope within the model.

One key feature of synchronous actions is that they can include assignment blocks. An example is given in Code Segment 3.13, where *tick* is the synchronizing action, and *clk* is being incremented when the action is executed. With this feature, variables across multiple parallel processes can be updated simultaneously, and the result is stored in a single state.

```
ExecuteA();
tick{= clk++ =};
ExecuteB();
```

Code Segment 3.13: Atomic Assignment During Synchronous Action

The final syntactical MODEST feature important to this work is the implementation of probabilistic choice. This is achieved using *palt*. The format is rather intuitive, and an example is shown in Code Segment 3.14. In this example, the distribution is discrete uniform, and MODEST has a shorthand equivalent for this assignment (see Code Segment 3.15).

```
palt{
  :(1/4): {= choice = 0 =}
  :(1/4): {= choice = 1 =}
  :(1/4): {= choice = 2 =}
  :(1/4): {= choice = 3 =}
}
```

Code Segment 3.14: Probabilistic Choice Example in MODEST

```
{= choice = DiscreteUniform(0, 4) =}
```

Code Segment 3.15: Probabilistic Choice Example in MODEST

3.5.2 Probabilistic Model Checking in Modest

Probabilistic model checking exhaustively searches the entire state space of a system in order to determine the likelihood of a particular state existing within that space. It can be both time-consuming and resource intensive, especially when checking systems with complex and concurrent behaviors. See Section 2.2 for more detail.

The MODEST TOOLSET includes a probabilistic model checker (*mcsta*), which builds the state space of a given probabilistic model in computer memory. MODEST, however, does not support certain temporal operators, such as $\bigcirc$ or $\Box$. While $\Box$ can be represented using the equivalent logic and operators that are available (i.e. $\neg\Diamond\neg x$), $\bigcirc$ cannot. The $\bigcirc$ operator is unique in that it represents the next state of the system, and requires an awareness of the position of one state in time relative to another (the current state and the next state). This makes it more complex, and is possibly the reason that the MODEST language does not yet support it.

3.5.3 Statistical Model Checking in Modest

Rather than use exhaustive techniques to search the state space of the system statistical model checking builds on Monte Carlo principals. This involves simulating the system, until the simulator can decide if the model satisfies some property. While this is not usually completely accurate, it can be done within a certain confidence threshold [37]. See Section 2.3 for more detail.

The primary tool available through the MODEST TOOLSET for probabilistic model verification is *modes*. This tool uses statistical methods to explore the state space of a model.

3.5.4 Modeling PSN Properties in Modest

Like [7,8], this work uses two main properties to determine the PSN of a system. These two properties are *resistiveNoise*, and *inductiveNoise* respectively. *resistiveNoise* occures when a router within the NoC experiences more than a specified threshold of activity in a given clock cycle. *inductiveNoise* is different in that it occurs when the difference in router activity between the last cycle and the current cycle is greater than a specified threshold. This threshold is a parameter of the system, and is discussed in greater detail in Section 6.3.

When a router experiences *resistiveNoise*, it increments a counter for the entire NoC system. Similarly, *inductiveNoise* receives its own counter. Property 3.5 is therefor checking the probability that the *resistiveNoise* of the system (or of the counter previously mentioned), will eventually be greater than some threshold (K), within N clock cycles. Property 3.6 shows a similar property for determining the system's *inductiveNoise* probability within N clock cycles.

$$\mathbf{P}_{=?}(\Diamond^{[\text{accumulate}(clk)\leq N]}\ resistiveNoise \geq K) \tag{3.5}$$

$$\mathbf{P}_{=?}(\Diamond^{[\text{accumulate}(clk)\leq N]}\ inductiveNoise \geq K) \tag{3.6}$$

CHAPTER 4
MOTIVATION

PSN in the power delivery network of a NoC is created by simultaneous switching of logic devices, causing a drop in the effective power supply voltage. It can be broken down into two major contributors, namely *resistiveNoise* and *inductiveNoise*. The *resistiveNoise* of a system is the product of both the current drawn and the total resistance of the NoC circuitry. Similarly, *inductiveNoise* is caused by the total inductance of the circuit as the current drawn changes. These properties together are used to quantify the magnitude of PSN within a NoC system. Recent work has also shown that the problem of PSN is only aggravated by the ongoing shift of CMOS transistor scaling: In an 8×8 NoC, scaling from 32-nm technology to 14-nm technology, resulted in the PSN amplitude increasing from 40% of the supply voltage to around 80% [38]. At a higher level, it is evident that heavy network traffic is a major contributing factor to PSN [38]. As a result of heavy network traffic, router supply voltages may become unpredictable, timing errors can occur, and network packets can be lost or corrupted. This results in degraded system performance. While non-probabilistic NoC verification can provide guarantees about the functional correctness of these systems, it cannot formally quantify the potential risk of PSN under random synthetic traffic patterns. Probabilistic verification however, can provide such information.

By analyzing patterns of PSN across specific chip designs, improved implementations can be developed, which distribute packet routing more efficiently; in order to mitigate PSN and its effects. In this work, flit injection is uniformly distributed across the system, meaning that each router will produce an equal number of flits for all other routers in the network. This pattern is generic enough to evaluate PSN, compared to particular real-world packet generation patterns. Thus, by simulating uniform traffic patterns, the resulting data should reveal where PSN is more likely to naturally occur in a network. Using this information, recommendations can be made to improve chip design, by moving heavy traffic away from areas of the network which naturaly experience higher PSN.

The work of [7] recommends moderating flit injection for a 2×2 NoC. In this work, the recommended pattern is applied: flits are generated the first three out of every ten clock cycles. Not only will this produce results comparable to those found in previous works (a requirement for the work done in Chapter 8), but it will also greatly improve simulation times [7, 8].

CHAPTER 5
MODULAR DESIGN FOR FORMAL NOC MODEL

This work focuses on developing a modular formal model for NoC verification. A major advantage to this models' modularity is its ability to be scaled and customized with minimal effort. It is this modularity which enables it to be adapted for applications of varying topological size. Additional configurations which can be easily changed within the model are discussed in Chapter 6. In order to keep the modular model similar to previous works [7,8], the same buffer size of 4 is used here, so that results can be compared to those generated from the monolithic designs. This will become important in Chapter 8.

The NoC router model presented in this chapter is synchronous in nature. That is, all routers in the model execute in parallel; synchronizing flit generation, buffer updates, and arbitration.

5.1 NoC Design Implementation

To accomplish this work, a single universal Modest router model was developed, as shown in Fig. 5.1. This router has four input buffers, as well as four output channels for interfacing with neighboring routers. It also contains a local input buffer for generating new flits. This is the only way by which new flits are introduced into the NoC. When instantiated, the router process is given its own unique ID, as well as the IDs of its neighbors. Disconnected routers are assigned an ID value of -1, to indicate that they do not have a neighbor on the given channel. This universal router design allows easy instantiation and configuration of routers with different locations, e.g., corner, edge, or central, within an arbitrary mesh network. Code Segment 5.1 shows this universal router being instantiated four times into a 2×2 mesh topology, and Fig. 5.2 shows the resulting network. The order of parameters for the *RouterBehavior* process is: this router's ID, the north neighbor's ID, the west neighbor's ID, the east neighbor's ID, and the south neighbor's ID.

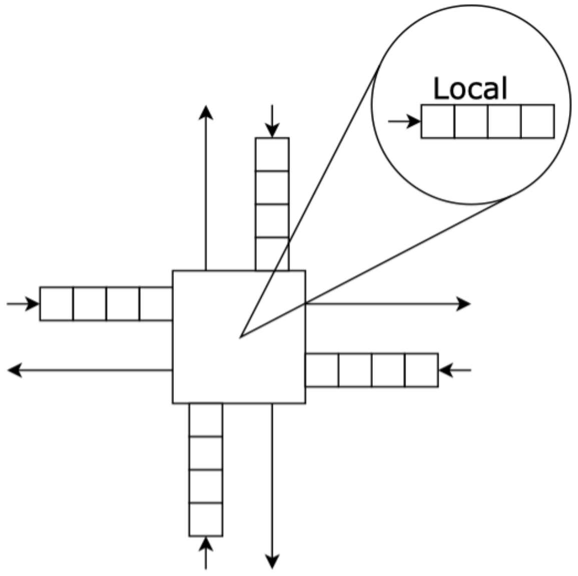


Fig. 5.1: Universal Router Diagram

```
par{
    :: Clock()
    :: RouterBehavior(0, -1, -1, 1, 2)
    :: RouterBehavior(1, -1, 0, -1, 3)
    :: RouterBehavior(2, 0, -1, 3, -1)
    :: RouterBehavior(3, 1, 2, -1, -1)
}
```

Code Segment 5.1: Parallel Processes of Routers for a Modular 2×2 NoC

An important component to the modular design, is the *Clock* process (shown in Code Segment 5.2). The clock variable, *clk*, essentially counts the number of cycles lapsed. This process is called recursively, and exits when a predetermined threshold, *CLK_UPPER*, is reached. The design for the universal router includes a similar exit procedure (see Code Segment 5.3). The use of the *tick* action once every cycle, is what keeps this process in sync with all routers in the NoC. As mentioned in Chapter 4, this process is necessary, because MODEST is an asynchronous language, and the NoC is a synchronous system.

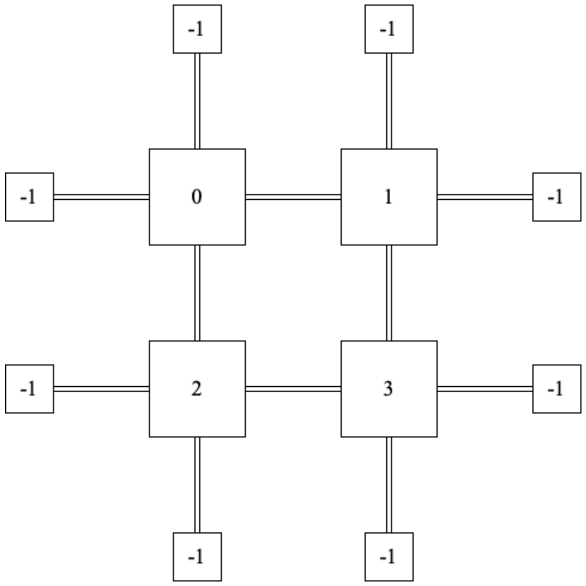


Fig. 5.2: NoC Neighbor Assignments for 2 × 2 Configuration. Buffers are not shown for each router.

```
process Clock(){
    tick{= clk++ =};
    if (clk <= CLK_UPPER) {
        Clock()
    }
    else {
        stop
    }
}
```

Code Segment 5.2: Clock Processes for a Modular NoC

In order to simplify the modular design, the universal router, is divided into five major sub-processes: Generating new flits on the local buffer, preparing the router for flit advancement, advancing flits to their respective output channels, updating the priority list, and updating the variables that track both *inductiveNoise* and *resistiveNoise*. A top module implements all sub-processes and describes the sequence of operations. Code Segment 5.3 shows this implementation. The *tick* and *tock* properties are custom actions, used to synchronize all parallel instantiations of this process. Additionally, the loop shown in Code Segment 5.3 includes a conditional statement for exiting the process once the desired number of clock cycles is reached. Notice the assignments being made to the *state* variable during these actions. As stated in Section 3.5.1, Modest allows for variable assignment during actions. All sub-processes execute this assignment to their respective *state* variable simultaneously. This is important for the verification of Property 7.6, which states that all sub-processes must remain in sync.

```
do{
    tick{= states[id] = 1 =};
    GenerateFlits(id);
    PrepRouter(id);
    tock{= states[id] = 2 =};
    AdvanceRouter(id);
    UpdatePiority(id);
    UpdateNoiseTracking(id);
    if(clk >= CLK_UPPER){
        stop
    }
    else{
        tau
    }
}
```

Code Segment 5.3: *RouterBehavior* Main Sequence Loop

5.1.1 Flit Generation

As mentioned in Chapter 4, this work is interested in the traffic patterns described in [7]. That is, flit generation is uniform, and only occurs in the first 3 out of every 10 clock cycles. The behavior for generating flits is described in the *GenerateFlits* sub-process, shown in Code Segment 5.4. A value is generated between 0 and one less than the maximum NoC ID. From there, the value is enqueued into the current router, increasing it by 1 if it is either equal to or greater than the current routers ID. In this way, the uniform distribution of flit generation to all other routers in the network is preserved, and the router will never generate a flit for itself (see Property 7.2). This model also allows for the frequency and distribution of flit generation to be configured, as described in Section 6.1.

```
int destination;
{=
    destination = DiscreteUniform(0, NOC_MAX_ID - 1)
=};
if(destination >= id){{=
    noc[id].channels[LOCAL].buffer = enqueue(destination + 1,
        noc[id].channels[LOCAL].buffer)
=}}
else{{=
    noc[id].channels[LOCAL].buffer = enqueue(destination,
        noc[id].channels[LOCAL].buffer)
=}}
```

Code Segment 5.4: Uniform Flit Generation for a Modular NoC Model

5.1.2 Flit Propagation

While parallel composition eliminates the need for redundant code, it also results in synchronization problems between routers, not previously relevant to other works [7, 8]. Primarily, the individual routers of the system are able to fall out of sync with one another, allowing for a single router to execute multiple cycles before its neighbors. Figure 5.3 shows an example of what can happen if the *tick* action for r_0 is allowed to go out of sync with the rest of the system. In this example, r_0 executes three full clock cycles, generating flits for each of the other routers in the network. This all happens before the other router processes even begin their first cycles.

Synchronizing the system once every cycle still allows for a router to write to its neighbors before those neighbors have started their cycles. This results in a write-before-read conflict, where flits are advanced through multiple routers in a single cycle, if those routers'

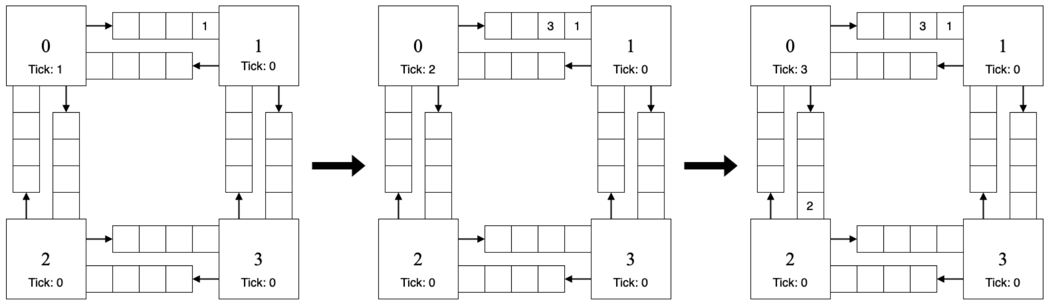


Fig. 5.3: Synchronization Conflict for 2 × 2 NoC due to the absence of synchronization actions.

input buffers were previously empty. Figure 5.4 shows an example of what can happen if the *tick* action is working, but the *tock* action is out of sync for all routers. In a single cycle, a flit is generated by r_0, forwarded to r_1 before being forwarded to r_3, and eventually gets absorbed by r_3.

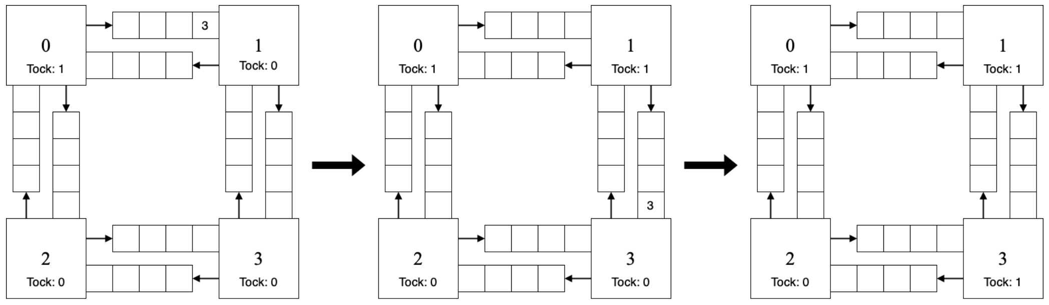


Fig. 5.4: Synchronization Conflict for 2 × 2 NoC. Due to Having no *tock* Action.

The sub-process *PrepRouter* was created to resolve the aforementioned incorrect write-before-read issue. Without it, the *tock* action is not a sufficient solution, because the *AdvanceRouter* sub-process still executes asynchronously between routers. To solve this problem, the *PrepRouter* sub-processes raises a flag for each of the buffers in a router, only if that buffer contains a flit. Then, the *tock* action synchronize the routers. This synchronization ensures that no router can be writing to its flags, after neighboring routers have started advancing flits into its respective buffers. During the *AdvanceRouter* sub-process, rather than check if its buffers are empty or not, the router checks these flags. A flit will only be forward from that buffer if its flag is set, and the flag will only be set if there was a flit present before the *PrepRouter* process was run. All flags are reset at the end of each cycle. Because buffers can only route one flit per cycle, only the flit which previously raised the flag will be forwarded on, while flits being added by neighbors during that time will not. Additionally, while router channels can only be used once in a single cycle, multiple flits can be removed from a router in a single cycle, if they have reached their destinations, and are next to be serviced within their respective buffers.

This latter design choice is not in accordance with the works of [7] or [8], both of which implement a design which allow for only one flit to be removed per router every cycle. This decision came about as the result of discrepancies between model results, which lead to a conversation with the main author of [38]. While both designs are valid, the reality is that the overarching hardware which feeds the NoC is unknown. It is therefore assumed in this work, that the hardware feeding into the NoC is capable of consuming multiple flits per cycle. Both the forwarding of flits, and the removal of flits occur within the *AdvanceRouter* sub-process.

5.1.3 Priority Tracking and Updates

During routing, flags are set when buffers are unable to be serviced due to conflicts. In the *UpdatePriority* sub-process, these flags are then used to generate the priority list for the next cycle. The implementation of this is shown in Code Segment 5.5. The process shown is called five times for each router, using the iterating index i, to advance through every item in the respective routers' priority list (*priority_list*). The updates to the priority list are stored in a temporary list (*priority_list_temp*), which is written to the parent priority list at the end of each cycle.

```
if(noc[id].channels[noc[id].priority_list[i]].serviced){
   {=
      0: noc[id].priority_list_temp[noc[id].total_unserviced
         + noc[id].serviced_index] = noc[id].priority_list[i],
      1: noc[id].serviced_index++
   =}
}
else{
   {=
      0: noc[id].priority_list_temp[noc[id].unserviced_index]
         = noc[id].priority_list[i],
      1: noc[id].unserviced_index++
   =}
}
```

Code Segment 5.5: Routine for Updating Priority Lists

In Section 3.1, a scenario was given, describing the priority updates illustrated in Fig. 3.1. In this example, there is a conflict between the south and local buffers in step (a). As a result, the *total_unserviced* property would be incremented to 1. This property, as well as the *serviced_index* and *unserviced_index* of each buffer, are set to 0 at the beginning of each clock cycle. As the process described in Code Segment 5.5 iterates through the items in the *priority_list*, it first checks the local buffer. This buffer is marked as serviced, so the buffer is placed at the $total\_unserviced + serviced\_index = 1$ position of the *priority_list_temp*. The *serviced_index* is then incremented to the value of 1. This process repeats for each buffer in the priority list, until the south buffer is reached. This is the first unserviced buffer in the *priority_list*. Because of this, it is placed at the $unserviced\_index = 0$ position of the *priority_list_temp*, and the *unserviced_index* is incramented by 1. This places the south buffer at highest priority for the next round. Once all items in the *priority_list* have been moved, the *priority_list_temp* is then copied into the *priority_list*, setting the priority order of buffers for the next cycle.

It should be noted, that on the first cycle, the priority list of each router is arbitrarily set to contain the values: north, east, south, west, and local, in that order. All permutations of this list where tested, and it was concluded that the order does not matter in regard to calculating PSN.

5.1.4 Noise Tracking

In the initial modular design, two additional flags are set during routing for helping to track *resistiveNoise* and *inductiveNoise*. One flag (*isResistive*), indicates if a buffer is not empty and is serviced, or has no neighbor. The other flag (*isInductive*), indicates that a buffer is empty, or is not serviced. Two additional flags (*wasResistive* and *wasInductive*), indicate the respective states of these flags from the last cycle. The *UpdateNoiseTracking* sub-process uses these flags to increment the *resistiveNoise* and *inductiveNoise* of the system. If a router's *isResistive* flag is set to true in a cycle, then *resistiveNoise* is incremented by 1. Updating *inductiveNoise* is more involved. If the system's *wasResistive* flag held in the last cycle and the *isInductive* flag holds in the current cycle, or if the system's *wasInductive* flag held in the last cycle and the *isResistive* flag holds in the current cycle, then *inductiveNoise* is incremented by 1. This is true for all routers, so in any given cycle, *resistiveNoise* and *inductiveNoise* can potentially increase by the amount equal to the total number of routers. Note that this is not the implementation used in the final model, as it does not scale with the system. Section 6.2 explains the issues in greater detail, as well as the changes made to resolve them.

CHAPTER 6
SCALING OF MODULAR NOC

6.1 Scaling the Modular NoC

The modular model presented in Chapter 5 is extremely flexible, and scaling it takes minimal effort. It is important to define what it means to be scaled with minimal effort. Using the 2×2 model as a base, any $n \times n$ model can be derived with only two simple modifications to the code. It must also be noted that for the model to function properly, the IDs of the routers must follow the following rules: each ID must be unique, and assignment of IDs starts at 0 and must be done incrementally (left to right and top to bottom). Fig. 5.2 and Fig. 6.1 demonstrate proper ID assignments for 2×2 and 3×3 NoCs, respectively.

The first step in scaling the modular NoC is to modify the number of routers in the NoC, as well as, the dimension of the NoC. Both can be done using a single variable. The maximum router ID is 1 - the total number of routers. Because this work focuses on only modeling $n \times n$ mesh topologies, it follows that $\sqrt{1 + NOC\_MAX\_ID}$ must always be a whole number, and is equal to the width and height of the system. This parameter is set in Code Segment 6.1.

```
const int NOC_MAX_ID = 3;
```

Code Segment 6.1: Max ID Declaration for a Modular 2×2 NoC

The second step in scaling the modular NoC is to instantiate a Modest process for each of the routers. The implementation of this for a 3×3 NoC is shown in Code Segment 6.2, and the resulting network is shown in Fig. 6.1. Each router takes its own ID, as well as the IDs of its immediate neighbors as parameters. As mentioned in Chapter 5, disconnected neighbors are assigned an ID of -1. The order of these parameters is: the current router ID, the north neighbor's ID, the west neighbor's ID, the east neighbor's ID, and the south neighbor's ID.

```
par{
    :: Clock()
    :: RouterBehavior(0, -1, -1, 1, 3)
    :: RouterBehavior(1, -1, 0, 2, 4)
    :: RouterBehavior(2, -1, 1, -1, 5)
    :: RouterBehavior(3, 0, -1, 4, 6)
    :: RouterBehavior(4, 1, 3, 5, 7)
    :: RouterBehavior(5, 2, 4, -1, 8)
    :: RouterBehavior(6, 3, -1, 7, -1)
    :: RouterBehavior(7, 4, 6, 8, -1)
    :: RouterBehavior(8, 5, 7, -1, -1)
}
```

Code Segment 6.2: Parallel Processes of Routers for a Modular 3×3 NoC

6.2 Counting *resistiveNoise* and *inductiveNoise* in Scaled Models

In previous works, modeling and analysis was only performed on 2×2 NoC models. This topology is made up entirely of corner routers. In implementing the 3×3 model shown in Fig. 6.1, for example, there are additionally four edge routers (r_1, r_3, r_5, r_7), and one central router (r_4). Edge routers have one more assigned neighbor than corners, and center routers have two more assigned neighbors than corners. This results in questions about how *resistiveNoise* and *inductiveNoise* are to be calculated based on the different router locations. In [38], it is explained that there is some arbitrary threshold of activity which must be decided upon, at which point *resistiveNoise* and *inductiveNoise* become probable. Working with the 2×2 topology, it was assumed that for there to be an increment in *resistiveNoise*, all buffers had to be serviced in a given cycle. Because corner routers only have three assigned input buffers, the noise threshold of three was implicitly being applied.

resistiveNoise is rather straight forward to calculate with a noise threshold implementation. So long as the number of buffers serviced in a cycle is equal to or greater than the noise threshold, *resistiveNoise* is incremented by 1 for that router. The difference with

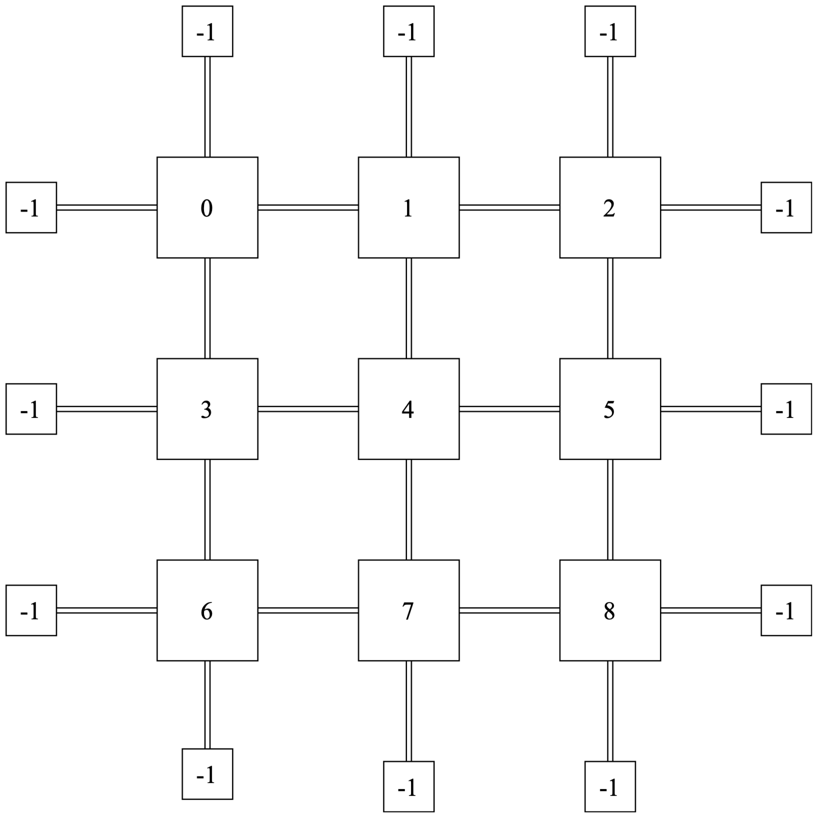


Fig. 6.1: NoC Neighbor Assignments for 3 × 3 Configuration. Buffers are not shown for each router.

inductiveNoise, is that it is not calculated based on a level of activity, but rather a change in activity levels between two consecutive cycles. Therefore, if the difference between a router's activity in the current cycle and the previous cycle is greater than the noise threshold, then *inductiveNoise* is incremented by one for that router.

In order to implement a noise threshold in the modular NoC model, the four flags used for tracking *resistiveNoise* and *inductiveNoise* (as described in Section 5.1.4) were replaced with two integer variables. The first (*thisActivity*), is used to track how many buffers have been serviced by the current router in the current cycle. The second (*lastActivity*), is used to track how many buffers were serviced by the current router in the last cycle. Using these two parameters, the *resistiveNoise* can be updated as shown in Code Segment 6.3. Similarly, the *inductiveNoise* can be updated as shown in Code Segment 6.4. Code Segment 6.4 also, shows how *inductiveNoise* is calculated, being the difference in noise between two cycles. Note that *resistiveNoise* and *inductiveNoise* are global variables of the NoC. The *ACTIVITY_THRESH* is a customizable parameter, which is assigned in Code Segment 6.5.

```
if(noc[id].thisActivity >= ACTIVITY_THRESH){
    {= resistiveNoise++ =}
}
else{
    tau
};
```

Code Segment 6.3: Routine for Incrementing *resistiveNoise*

```
if (abs(noc[id].lastActivity - noc[id].thisActivity) >= ACTIVITY_THRESH) {
    {= inductiveNoise++ =}
}
else{
    tau
};
```

Code Segment 6.4: Routine for Incrementing *inductiveNoise*

```
ACTIVITY_THRESH = 3;
```

Code Segment 6.5: NoC Activity Threshold Customization

6.3 NoC Customization

The model designed in this work also allows for additional customizations to be made, which go beyond the scope of scalability. The first of the customizations, is changing the length of the router input buffers. This modification can be made in the code where the lines appear from Code Segment 6.6. Any size can be specified, depending on the needs of the application.

```
const int BUFFER_LENGTH = 4;
```

Code Segment 6.6: Router Buffer Length Customization

As mentioned in Section 6.2, the dynamics of counting *resistiveNoise* and *inductiveNoise* changes with NoCs which contain more than just corner routers. The *ACTIVITY_THRESH* variable, shown in Code Segment 6.5 can be set to indicate how much activity (or difference in activity) is needed, before a router will contribute to the total PSN of the system. For example, setting the *ACTIVITY_THRESH* to 2, will cause routers to contribute to the systems *resistiveNoise*, if two or more buffers are serviced in a cycle.

Additionally, the flit injection rate can be changed. For this work, the flit injection pattern is set to generate flits during the first three of every ten cycles. This modification can be made in the code where the lines appear from Code Segment 6.7. As shown, the *INJECTION_RATE_NUMERATOR* indicates how many cycles in which flits will be generated, within every *INJECTION_RATE_DENOMINATOR* cycles. The pattern used in this work was chosen to match that of [7,8].

```
const int INJECTION_RATE_NUMERATOR = 3;
const int INJECTION_RATE_DENOMINATOR = 10;
```

Code Segment 6.7: Injection Rate Customization

While this work focuses on uniform flit generation patterns, the model allows for any flit injection pattern to be represented. By checking the ID of the current router, the flit injection patterns can even be specified by router through the use of conditional statements. Further, the global *clk* variable (specified in Code Segment 5.2), can be used to dynamically change the flit injection pattern based on time. Code Segment 6.8 demonstrates an arbitrary example of custom flit generation for a 2×2 NoC. In this example, while under 20 clock cycles, only r_3 is active. Once above 19 clock cycles, routers r_0, r_1, and r_2, each begin producing flits with uniform distribution, destined to all other routers in the network. While under 100 clock cycles, a third of the flits generated by r_3 are destined for r_0, and the rest are destined for r_2. After 100 clock cycles, all routers produce a uniform distribution of flits. Such control over flit generation patterns allows for this modular NoC to be tailored to a broader scope of applications.

```
int destination;
if(clk < 100 && id == 3){
    palt{
        :(1/3): {= destination = 0 =}
        :(2/3): {= destination = 2 =}
    }
}
else if(clk < 20){
    tau
}
else{
    palt{
        destination = DiscreteUniform(0, NOC_MAX_ID - 1)
    }
};
if(destination >= id){{=
    noc[id].channels[LOCAL].buffer = enqueue(destination + 1,
        noc[id].channels[LOCAL].buffer)
=}}
else{{=
    noc[id].channels[LOCAL].buffer = enqueue(destination,
        noc[id].channels[LOCAL].buffer)
=}}
```

Code Segment 6.8: Custom Flit Generation for a Modular 2×2 NoC

CHAPTER 7
FORMAL VERIFICATION OF NOC

In this chapter, the functional correctness of the modular NoC model is verified. While this model seeks to mitigate complexity by promoting the reuse of otherwise redundant code, some complexities remain, which make it difficult to debug. Such debugging stems from discrepancies between the expected *resistiveNoise* or *inductiveNoise*, and the actual results. In order to simplify this process, the work of verifying the NoC is divided into three categories: router verification, inter-router verification, and composed system verification. This helps to narrow down the cause of errors. In this chapter the term process refers to the process construct found in the Modest language, which are the building block of Modest model behavior.

In this chapter a notation is introduced to express the behavior of the routers. Previously, routers have been expressed using the notation r_i, where i is the ID of the router. Here, the addition of a superscript is given, to indicate the buffer being addressed. The script becomes r_i^b, where i is still the router ID, and b is the buffer index. This index can be specific (i.e. N, S, E, W, L), indicating the north, south, west, east, or local buffer, or in can be generalized as *source* or *destination*. The next notation is $j \twoheadrightarrow r_i^L$. Here the local buffer of router i is generating flit j, where j is the ID of the destination router. The final notation introduced here is $r_i^{source} \xrightarrow{x} r_j^{destination}$. Here r_i is sending a flit with the destination router ID of x from its own *source* buffer, to the *destination* buffer of r_j.

The Modest language does not support iterating through values of variables within properties, so the properties given in this chapter had to be encoded in all of their variants. For example, the property implemented in Code Segment 7.1 was encoded twelve times, in order to cover every permutation of i and j in a 2×2 network. This also means that the implementations of these properties would need to be scaled with the model. This could become quite cumbersome, as each property increases the time it takes to check the model.

Notice in the code of this chapter, that the *Pmax()* method returns a probability. By evaluating the result using comparative operators (e.g. $<$, $>$, $==$), a boolean property is created, which the Modest Toolset will evaluate to either *True* or *False*.

7.1 Router Verification

This section focuses on verifying the behavior of the processes in which routers only access their own resources, and are independent of their neighbors. That is, processes in which routers are not accessing the buffers of other routers. Such independent processes include generating flits and updating the priority list.

Each router must eventually generate a flit for every other router in the system. While some applications might experience little to no traffic between a particular pair of routers, this work in focused on the effects of PSN, assuming uniform random flit generation for each router. Because of this, there should eventually be communication between every pair of routers. The encoding for Property 7.1 is shown in Code Segment 7.1.

$$\forall i : \forall j \neq i : \Diamond(j \twoheadrightarrow r_i^L) \tag{7.1}$$

```
property p = Pmax(<>(peekFront(noc[i].channels[LOCAL].buffer) == j)) > 0;
```

Code Segment 7.1: Encoding of Property 7.1

Routers must not generate flits for themselves. There is no readily evident need for a router to send a flit to itself, as any hardware connected to that router would already have access to such information – itself being the sender. This model seeks to avoid such a scenario entirely. For this property (represented by Property 7.2), The encoding for Property 7.2 is shown in Code Segment 7.2.

$$\forall i : \neg\Diamond(i \twoheadrightarrow r_i^L) \tag{7.2}$$

```
property p = Pmax(<>(peekFront(noc[i].channels[LOCAL].buffer) == i)) == 0;
```

Code Segment 7.2: Encoding of Property 7.2

All buffers must always exist in the priority list. The purpose of this property is to verify that the priority list is not becoming corrupted, by ensuring that channels are not being removed nor duplicated. Without this property, channels may be removed from the priority list, resulting in that channel being starved (never able to send). In this model, the priority lists of all routers are arbitrarily initialized to (*north*, *east*, *south*, *west*, *local*). Because the size of the priority lists is also never changing, it follows that there are no duplicate items in the priority list. Duplicate items would result in a biased advantage for certain channels, resulting in skewed *resistiveNoise* and *inductiveNoise* values. The process of updating the priority list is described in more detail in Section 5.1. The encoding for Property 7.3 is shown in Code Segment 7.3. Notice that in the code, in addition to i being a placeholder for the router ID, b is a placeholder for the buffer index (i.e. NORTH, EAST, SOUTH, WEST, LOCAL). Meaning this property has twenty variants for verifying a 2×2 network.

$$\begin{aligned}
\forall i : \Box(&local \in r_i.priorityList \\
\wedge &north \in r_i.priorityList \\
\wedge &east \in r_i.priorityList \\
\wedge &south \in r_i.priorityList \\
\wedge &west \in r_i.priorityList)
\end{aligned} \tag{7.3}$$

```
property p = Pmax(<>(!(
    noc[i].priority_list[0] == b || noc[i].priority_list[1] == b ||
    noc[i].priority_list[2] == b || noc[i].priority_list[3] == b ||
    noc[i].priority_list[4] == b))) == 0;
```

Code Segment 7.3: Encoding of Property 7.3

7.2 Inter-Router Verification

This section focuses on verifying the behavior of processes which coordinate the interactions between routers. Such processes are involved in the passing of flits from one router to another, and manipulate only the two routers involved in the transaction. A binary function (*neighbors()*), is therefore defined, which takes two router IDs as parameters and only evaluates to true if the two routers identified are assigned to one another as neighboring routers. This neighbor assignment is outlined in Section 6.1.

A flit can only be routed when its source buffer is not empty and its destination buffer is not full. This property avoids the conflicts that might arise from input buffers becoming congested. Such a scenario can occur when an input buffer is competing with other buffers in the router repeatedly over many cycles. For example, in the 3×3 NoC from Fig. 6.1, routers r_0 and r_2 might both generate flits destined for r_4, repeatedly over the course of many cycles. Here, both r_0 and r_2 are competing for the south channel of r_1, and they will each only get priority once every other cycle. Thus, the east and west buffers of r_1 will begin to fill. The solution is to never allow forwarding a flit to a full input buffer. This means keeping it in the senders buffer, and marking that buffer as unserviced. This does not completely eliminate the problem however, as flits will continue to be generated, and other buffers will become more congested. Over enough time, the local buffers will also become congested. This design is implemented such that if any local buffer becomes full, that router will halt generating flits until space on the local buffer becomes available. The function *isEmpty()* returns true only if the specified buffer contains no flits. The function *isFull()* returns true only if the specified buffer contains the maximum number of flits allowed. The encoding for Property 7.4 is shown in Code Segment 7.4. Notice in the code, that the buffers ($b1$ and $b2$) are placeholders for the buffer indexes.

$$\begin{aligned}\forall i : \forall j \neq i : \forall x : \Box(\text{neighbors}(i, j) \wedge r_i^{source} \xrightarrow{x} r_j^{destination} \\ \implies \neg(\text{isEmpty}(r_i^{source}) \vee \text{isFull}(r_j^{destination})))\end{aligned} \tag{7.4}$$

```
property p = Pmax(<>(!(!isRouting[i].routing[SOUTH] || (
    !noc[i].channels[SOUTH].isEmpty &&
    !noc[j].channels[NORTH].isFull)))) == 0;
```

Code Segment 7.4: Encoding of Property 7.4

This formula states that for every permutation of two unique routers (r_i and r_j), if they are neighbors, a flit (x) sent from r_i to r_j, implies that the source buffer of r_i is not empty, nor is the destination buffer of r_j full.

A flit can only be routed if the channel has not been used in the current cycle. Because the NoC of interest is synchronous, it operates on a centralized clock. It would be physically impossible – under the synchronous assumption – to send two flits on the same buffer in a single cycle. This property reinforces the realities of the physical hardware and architectures of actual NoCs. It is not unreasonable to assume that an NoC might only be able to send one flit per channel in any given cycle.

For this property (represented by Property 7.5), i and j are router IDs. While buffers are located at the inputs to the routers, channels are the outputs through which the flits must pass to reach the neighboring routers' input buffers. Each router contains a list of output channels. The function *channelTo* returns the channel which connects the parent router (r_i) to the router whose ID is being referenced (r_j). The function *isUsed* returns true if the given channel has not been used in this clock cycle. The encoding for Property 7.5 is shown in Code Segment 7.5. Notice in the code, that the buffer (b) is a placeholder for the buffer index.

$$\forall i : \forall j \neq i : \forall x : \Box(\text{neighbors}(i,j) \wedge r_i^{source} \xrightarrow{x} r_j^{destination} \\ \implies \neg\text{isUsed}(r_i.\text{channelTo}(j))) \tag{7.5}$$

```
property p = Pmax(<>(!(!isRouting[i].routing[b]
    || !noc[i].used[b]))) == 0;
```

Code Segment 7.5: Encoding of Property 7.5

This formula states that for every permutation of two unique routers (r_i and r_j), if they are neighbors, a flit (x) sent from r_i to r_j, implies that the channel between them has not been used in this cycle.

7.3 Composed System Verification

This section focuses on the broader property, which govern all routers in the network simultaneously. This property is used to ensure the synchronization of the entire system, and to help correctly quantify PSN.

All routers must synchronize between generating and sending flits. Using the model developed in this work, all routers in a composed NoC are simulated in parallel. Without synchronizing actions to coordinate the timing of routers, buffers could potentially run repeatedly in a single cycle, allowing flits to advance through multiple routers. This would result in much higher, and incorrect readings for both *resistiveNoise* and *inductiveNoise*.

For this property, the *state* variable is a binary variable, which holds the value of 0 when the router is in its flit generation state, and 1 when the router is in its flit forwarding state. The specific sub-processes which fall into these two categories are shown in Code Segment 5.3. This property guarantees that all routers will both generate, and then forward flits simultaneously. The encoding for Property 7.6 is shown in Code Segment 7.6.

$$\forall i : \forall j : \Box(r_i.\text{state} = r_j.\text{state}) \tag{7.6}$$

```
property p = Pmax(<>(!(states[i] == states[j]))) == 0;
```

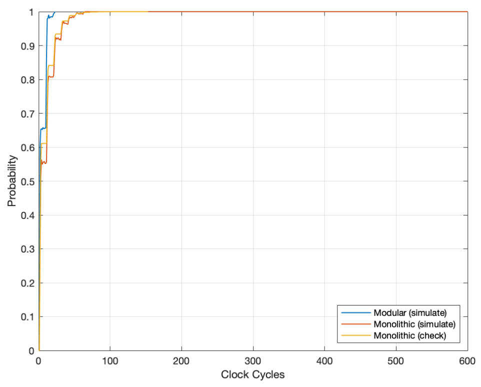
Code Segment 7.6: Encoding of Property 7.6

CHAPTER 8
COMPARATIVE CHECKING OF NOC

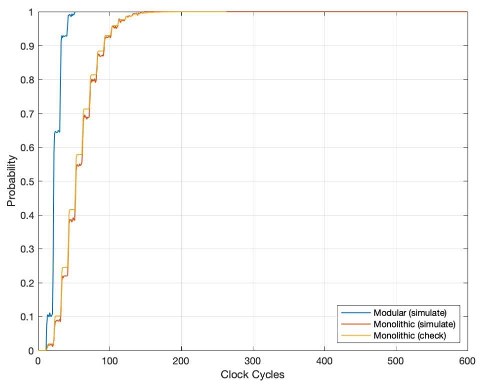
The NoC designed in this work attempts to be a close representation of real hardware, in which many of the mechanics of the model cannot be abstracted away. As a result, doing a model checking results in state explosion. It is for this reason that the results presented in this work was derived through statistical model checking. In order to give a more fair comparison, new results were generated using the models developed by [7] and [8], using the same statistical model checking tools.

Initial implementation of the modular design yielded data quite different from that observed in the works of [7] and [8]. These differences are shown in Fig. 8.1. Because the modular model had already been proven using the properties addressed in Chapter 7, it was concluded that either the previously developed models had errors, or that the architectural functionality of NoCs had been interpreted differently in the previous works. Comparative checking is a novel practice, which enables tracking discrepancies between two models, by comparing inputs and outputs, without the need to track the entire state space. This is important because, while the model developed in this work is modular, the models available for comparison were not. This resulted in many major architectural differences. As a result, comparing the states of the two would have been very laborious.

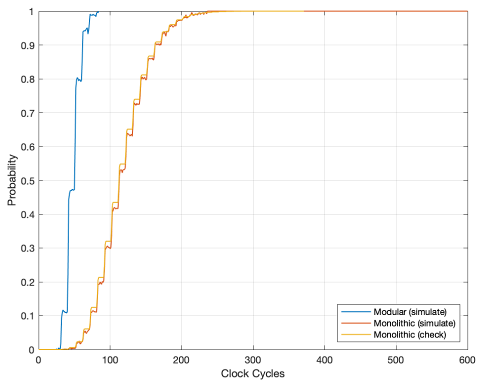
In order to have a fair comparison, a 2×2 NoC would have to be used, as the previous works could not be scaled without drastic modifications to their code. The implemented configuration of a 2×2 modular model is shown in Fig. 5.2. For reference, a situation in which a router is expected to increase the *resistiveNoise* of the system, and is ultimately observed doing so, will be referred to as a true positive. Likewise, a situation in which a router is expected not to increase the *resistiveNoise* of the system, and is ultimately observed behaving as expected, will be referred to as a true negative. A situation in which a router is expected to increase the *resistiveNoise* of the system, and is ultimately observed not



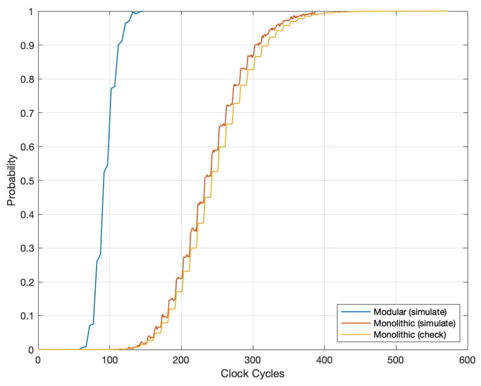
(a) $resistiveNoise \geq 1$



(b) $resistiveNoise \geq 5$



(c) $resistiveNoise \geq 10$



(d) $resistiveNoise \geq 20$

Fig. 8.1: Initial Comparison of *resistiveNoise* in 2×2 NoC Models

behaving as expected, will be referred to as a false negative. Lastly, a situation in which a router is expected not to increase the *resistiveNoise* of the system, and is ultimately observed not behaving as expected will be referred to as a false positive.

If two NoC models yield contradicting results, it must be that there is a discrepancy in at least one of the two models. Either the model predicting the lesser amount of *resistiveNoise* is correct, or it contains an error which results in false negatives. Similarly, the model predicting the greater amount of *resistiveNoise* is either correct, or contains an error which results in false positives. Both models may contain errors, but both cannot be correct.

It is possible that one of the models contains many errors which both increase and decrease the results. In such a case the more dominant error would need to be resolved before the other could be identified. In the event that the two errors cancel one another out, and the resulting PSN levels are the same between models, this method will fail.

In order to track down the cause of the discrepancy, one must first assume that one of the models is correct, and then test the other against a set of predetermined flit generation patterns, in order to verify that it is calculating PSN correctly. If the model assumed to be faulty is seeing higher PSN than expected, then the routers in the network should be checked to make sure that they satisfy patterns which should result in true negatives. Likewise, if the model assumed to be faulty is seeing lower PSN than expected, then the routers in the network should be checked to make sure they satisfy patterns which should result in true positives. During these tests, if the model evaluates any members of the input set as either false positives or false negatives respectfully, then the model is disproven, and the necessary properties can be developed in order to correct its functionality.

It was determined that within two consecutive clock cycles, no *resistiveNoise* could occur. This is because it takes one clock cycles for the flits to generated, and another cycle for the flit to propagate to a neighboring router. As a result, for the first two clock cycles, routers only have their local buffers to service, and the rest are empty. Additionally, within three clock cycles, any router could send a flit to any other router in a 2×2 network, with the first cycle being used to generate the flit. For these reasons, the set of all patterns which should result in true positives (within three clock cycles), was used for testing the models. Table 8.1 shows the five possible patterns for r_1 of the NoC illustrated in Fig. 5.2, which should produce *resistiveNoise* within three clock cycles. It should be noted that while all routers must be verified individually in the previous works, only the one implementation of the generic router needs to be verified for the modular model. Here we introduce the notation $i \twoheadleftarrow r_i^b$, where r_i removes flit i from its own buffer b, as the flit has reached its destination.

Using the patterns shown in Fig. 8.1, and by assuming that the modular model was the one behaving correctly, the concrete model developed in [7] was then checked to verify that it satisfied all true positives which could occur within three clock cycles. By forcing

Case	Clk = 0	Clk = 1	Clk = 2
1	$1 \twoheadrightarrow r_0^L$, $1 \twoheadrightarrow r_3^L$	$r_0^L \xrightarrow{1} r_1^W$, $r_3^L \xrightarrow{1} r_1^S$, $0 \twoheadrightarrow r_1^L$	$1 \twoheadleftarrow r_1^W$, $1 \twoheadleftarrow r_1^S$, $r_1^L \xrightarrow{0} r_0^E$
2	$1 \twoheadrightarrow r_0^L$, $1 \twoheadrightarrow r_3^L$	$r_0^L \xrightarrow{1} r_1^W$, $r_3^L \xrightarrow{1} r_1^S$, $2 \twoheadrightarrow r_1^L$	$1 \twoheadleftarrow r_1^W$, $1 \twoheadleftarrow r_1^S$, $r_1^L \xrightarrow{2} r_0^E$
3	$1 \twoheadrightarrow r_0^L$, $1 \twoheadrightarrow r_3^L$	$r_0^L \xrightarrow{1} r_1^W$, $r_3^L \xrightarrow{1} r_1^S$, $3 \twoheadrightarrow r_1^L$	$1 \twoheadleftarrow r_1^W$, $1 \twoheadleftarrow r_1^S$, $r_1^L \xrightarrow{3} r_3^N$
4	$3 \twoheadrightarrow r_0^L$, $1 \twoheadrightarrow r_3^L$	$r_0^L \xrightarrow{3} r_1^W$, $r_3^L \xrightarrow{1} r_1^S$, $0 \twoheadrightarrow r_1^L$	$r_1^W \xrightarrow{3} r_3^N$, $1 \twoheadleftarrow r_1^S$, $r_1^L \xrightarrow{0} r_0^E$
5	$3 \twoheadrightarrow r_0^L$, $1 \twoheadrightarrow r_3^L$	$r_0^L \xrightarrow{3} r_1^W$, $r_3^L \xrightarrow{1} r_1^S$, $2 \twoheadrightarrow r_1^L$	$r_1^W \xrightarrow{3} r_3^N$, $1 \twoheadleftarrow r_1^S$, $r_1^L \xrightarrow{2} r_0^E$

Table 8.1: Patterns for Generating *resistiveNoise* in a 2×2 Model

the local buffers to inject certain patterns, a sort of testbench could be created for the model. Using the methods discussed in 6.3, these custom flit generation patterns could be implemented within the modular model. The code for implementing the first pattern is shown in Code Segment 8.1. The processes for forcing the previously developed model to inject specific patterns was much more extensive [7], and is not shown here.

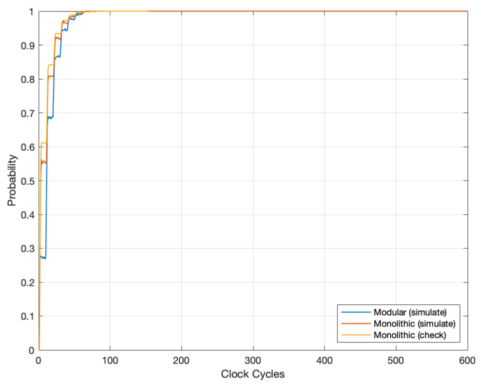
```
if(clk == 0 && (id == 0 || id == 3)){[=
    noc[id].channels[LOCAL].buffer = enqueue(1,
        noc[id].channels[LOCAL].buffer)
=]}
else if(clk == 1 && id == 1){[=
    noc[id].channels[LOCAL].buffer = enqueue(0,
        noc[id].channels[LOCAL].buffer)
=]}
```

Code Segment 8.1: Modular Implementation for Forcing *resistiveNoise* Pattern: Case 1

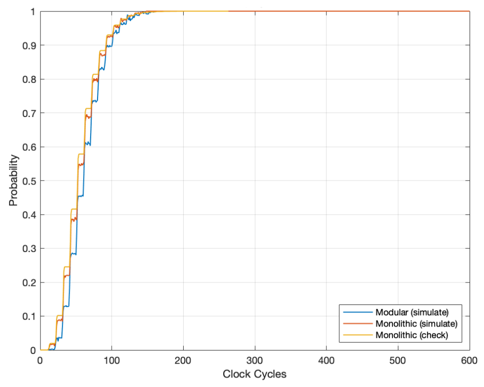
The concrete model produced in [7] only counts *resistiveNoise* for cases 4 and 5 of Table 8.1, while the modular model counted *resistiveNoise* for all cases (as expected). These results revealed that the previous model did not count *resistiveNoise* when r_1 received more than one flit destined for itself in a single cycle, because its architecture only allowed for it to handle the removal of one flit at a time. Further testing revealed that this was the case for all routers in the non-modular system.

After analyzing the design decisions made in [7], it was determined that the modular model was more in line with the NoC described in [38]. A conversation was held with the authors of the aforementioned works, and this observation was validated. In order to verify that this was the only discrepancy between the models, the necessary changes were made to the routing sub-process of the modular model, in order to match the previous architectures, and results were generated again for the 2×2 NoC. The results in Fig. 8.2 show how the three models are now much more similar regarding *resistiveNoise*. Results were also generated for *inductiveNoise* using the 2×2 models, and a comparison of these results are shown in Fig. 8.3. These results show that, besides the design decisions effecting the *resistiveNoise*, there were no other issues influencing the *inductiveNoise* of the modular system. Once these results where generated and the models were found to be equivalent, the modular made was reverted to its last iteration, and all further results presented in this work will be based on that design.

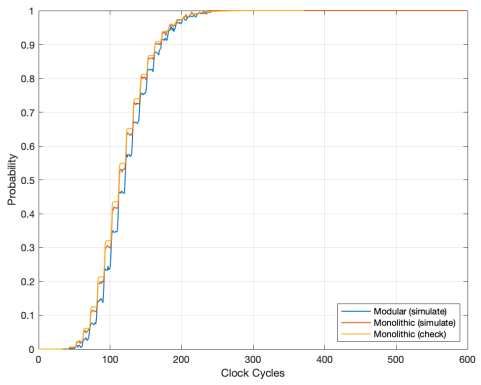
The properties developed in Chapter 7 not only helped to verify the functional behavior of the modular model, but held up against the discrepancy detected between this and previous works. As a result of that discrepancy, the comparative verification process developed in this chapter, was effective in helping to identify the source of the problem. It is worth noting that the comparative verification process is a results based approach, and is therefore only helpful in tracking issues which affect the results. When using simulation techniques, there may be discrepancies in the results which are not due to errors in the models, but rather, are inherent to simulation itself.



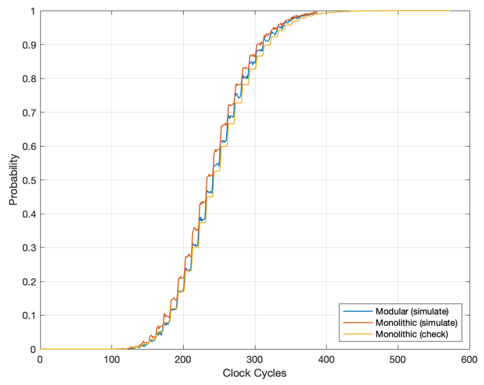
(a) $resistiveNoise \geq 1$



(b) $resistiveNoise \geq 5$

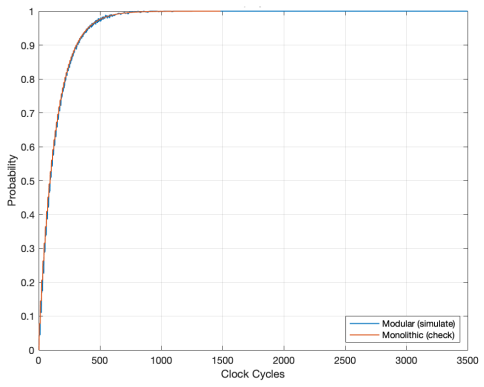


(c) $resistiveNoise \geq 10$

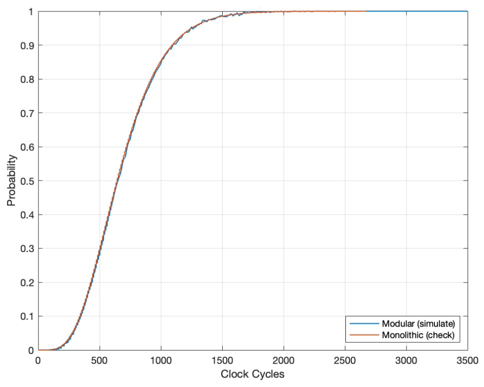


(d) $resistiveNoise \geq 20$

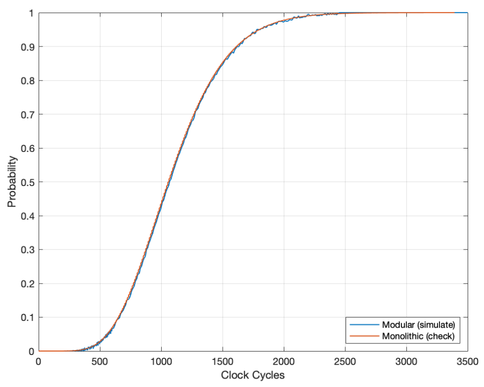
Fig. 8.2: Second Comparison of *resistiveNoise* in 2×2 NoC Models



(a) $inductiveNoise \geq 1$



(b) $inductiveNoise \geq 5$



(c) $inductiveNoise \geq 8$

Fig. 8.3: Final Comparison of *inductiveNoise* in 2×2 NoC Models

CHAPTER 9
RESULTS AND DISCUSSION

The results presented in this work were generated on a machine using a 12-core AMD Ryzen Threadripper Processor, operating at 3.5 GHz. The machine has 132 GB of memory, and runs Ubuntu Linux version 22.04 LTS. Additionally, the Modest Toolset version 3.1.182 was used. For simulations, all Modest parameters where default, except for *–max-run-length*, which was set to 0. Most of the results where generated in less than 24 hours, though some (more especially for *inductiveNoise*), took upwards of 5 days. This makes it relatively more intensive than the monolithic model, though only by a matter of hours at most.

While Chapter 8 shows the difference in PSN between this and previous works, Fig. 9.1 and Fig. 9.2 show the *resistiveNoise* and *inductiveNoise* for a 2×2 implementation of the modular model alone. This is with the new architecture presented in Chapter 8. Together, these figures show that PSN is much higher given the assumption that multiple flits can be removed by a router in a single cycle.

The advantage of a modular NoC design is the ability to scale it with ease. In order to verify the complete functionality of the generic modular router designed in this work, implementing it in a 3×3 NoC model would be ideal. This is because a 3×3 NoC is the smallest network size which utilizes edge, corner, and center nodes. As a comparable probabilistic 3×3 NoC model does not exist in published work for us to do comparative verification on, we have to assume that as our model is correct, as it does satisfy all properties from Chapter 7. Fig. 9.3 and 9.4 show the *resistiveNoise* and *inductiveNoise* respectively, for the scaled 3×3 NoC model. These figures show a very pronounced pattern, where there are steep slopes followed by rough steps. The roughness of the graphs is a result of using simulations tools. The default settings are being used to generate the data presented here, and the default value for the confidence parameter is 95%. The Modest Toolset can

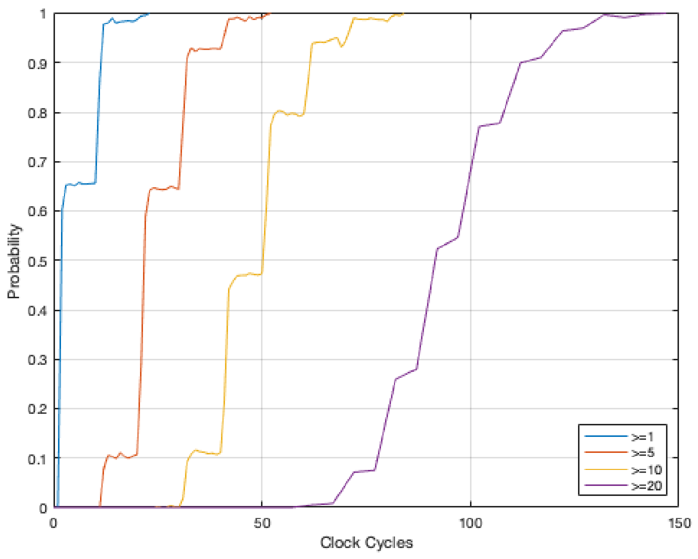


Fig. 9.1: *resistiveNoise* for Modular 2×2 Model

perform at higher confidence levels, though simulation takes much longer. The step pattern is a result of using three out of ten flit injection bursts. Only the first three clock cycles of every ten actually produce flits, and therefore effect the PSN of the system.

With the scaled 3×3 model working, additional information can be derived. Rather than allowing all routers to increment the system's *resistiveNoise* and *inductiveNoise*, a condition can be used to check just a set of routers. Code Segment 9.1, shows an implementation for checking only the PSN of router 0. This code is just a modification to Code Segment 6.4. Using this method, the PSN properties for routers 1, 3, and 4 were also verified. It is important to note that only these routers were simulated, because router 0 is a corner, and should experience the same PSN as routers 2, 6, and 8. Similarly, router 1 should be the same as 7, and 3 should be the same as 5. Perhaps a little less intuitive is the fact that the horizontal edge routers (i.e. 1 and 7), should not experience the same PSN and the vertical edges (i.e. 3 and 5). This is because X-Y routing is being used, and therefore horizontal edges should be experiencing more PSN than the edges. For reference see Fig. 6.1. The *resistiveNoise* results of this test are shown in Fig. 9.5, and the *inductiveNoise* results are shown in Fig. 9.6

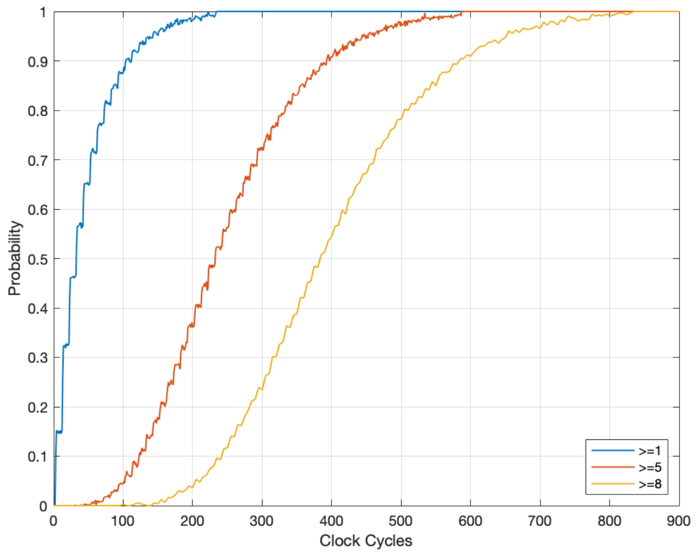


Fig. 9.2: *inductiveNoise* for Modular 2×2 Model

```
if (id == 0
    && abs(noc[id].lastActivity - noc[id].thisActivity) >= ACTIVITY_THRESH) {
        {= inductiveNoise++ =}
}
else{
    tau
};

if(id == 0 && noc[id].thisActivity >= ACTIVITY_THRESH){
    {= resistiveNoise++ =}
}
else{
    tau
};
```

Code Segment 9.1: Checking PSN for Router 0 Only

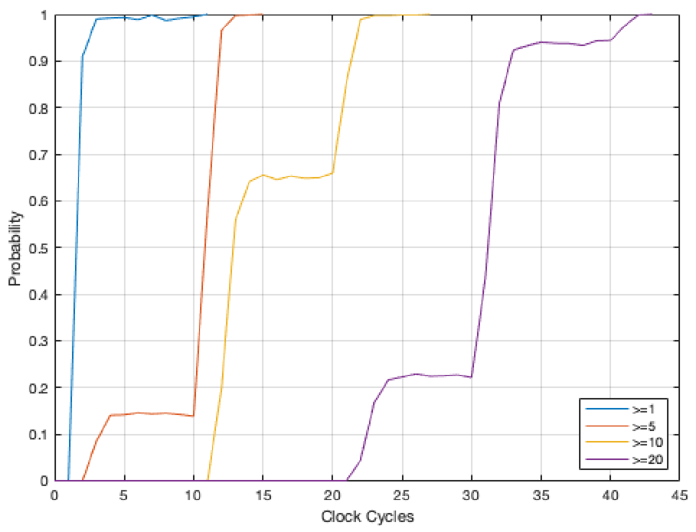


Fig. 9.3: *resistiveNoise* for Modular 3×3 Model

9.1 Discussion

Finally, the results given in this chapter show that the scaled 3×3 model yielded new and valuable information to the field of probabilistic NoC verification. In comparing the PSN properties between the 2×2 and 3×3 implementations, it can be seen that PSN scales proportionally with the size of the NoC. This observation is over the entire system, and localized results may be different.

The router specific results in Fig. 9.5 and 9.6 show that PSN increases towards the center of the router. While 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 could be in part due to the design of the NoC. For this design, the threshold for routers to experience *resistiveNoise* was set to three or more buffers being serviced in a cycle. Because, the center router has more active buffers than the other routers on the chip, it is intuitive that it should experience more PSN. Even so, the horizontal edge routers experienced higher PSN than the vertical edges, regardless of having the same number of buffers. This is likely a result of X-Y routing causing higher traffic through those areas. It can be concluded, that keeping the more traffic-prone neighbors on the perimeter of the NoC may be advantageous.

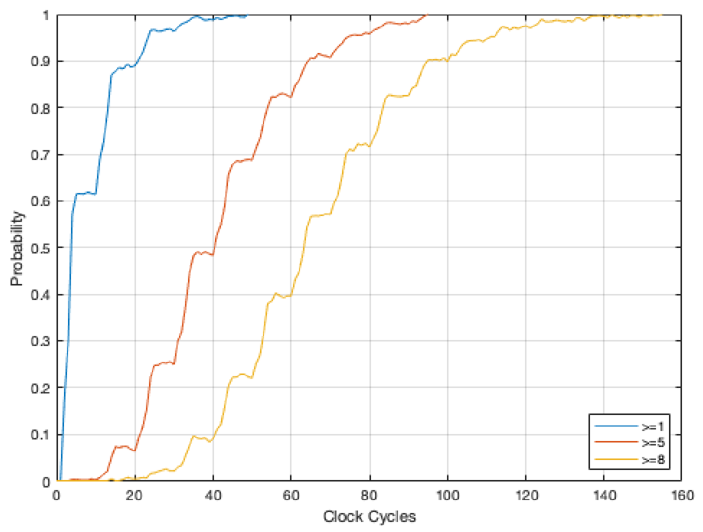


Fig. 9.4: *inductiveNoise* for Modular 3×3 Model

Previously, the largest topology achieved for a probabilistic NoC model was a 2×2 network. As mentioned, the routers in such a configuration would only have three input buffers. With the threshold for PSN being set to three serviced buffers, this inherently results in lower PSN across routers with higher conflict rates. This is because if even one buffer goes unserviced in a cycle, there can be no PSN on that router. At the same time, having conflicts results in lower router efficiency. This measure does not scale to larger NoCs, making it difficult to make any particular recommendation to chip designers based on these findings.

In the work of [7], it is concluded that by injecting delays between flits, PSN can be reduced. While this is true, this work affirms that PSN is difficult to completely eradicate using routing flit injection pattern alone. The larger an 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 being reserved for vertical neighbors. In this way, flits can propagate through the system 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.

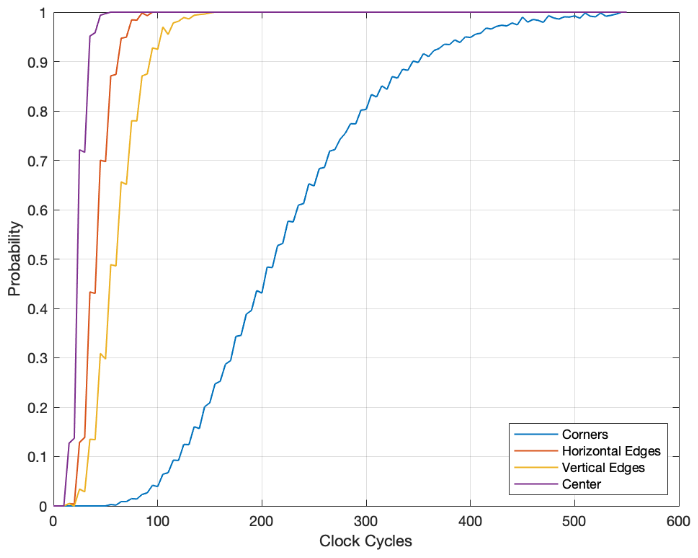


Fig. 9.5: Router Specific *resistiveNoise* for Modular 3×3 Model

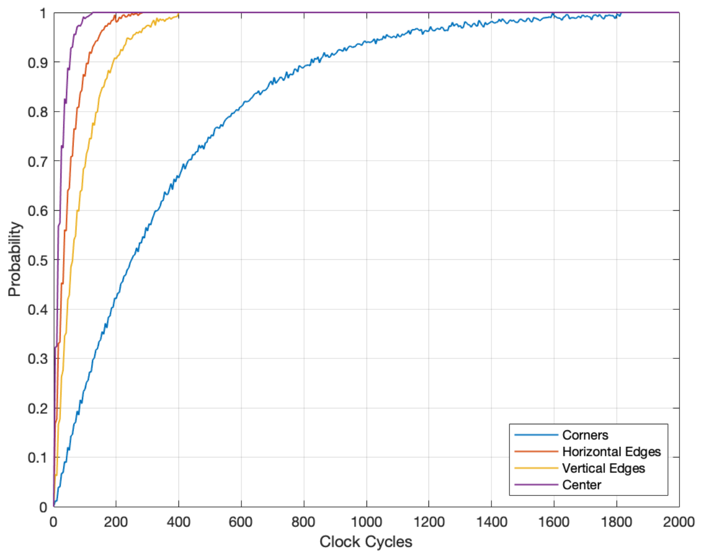


Fig. 9.6: Router Specific *inductiveNoise* for Modular 3×3 Model

CHAPTER 10
CONCLUSIONS

Better modeling of NoC systems will aid in resolving the challenges currently facing their advancement. Formally modeling and verifying such systems probabilistically will further help to develop a better understanding of characteristics such as PSN, and provide quantitative guarantees to aid in the validation of NoC design and behavior.

Having a modularized design will greatly assist in the development of such systems by providing a scalable architecture which can be customized to a range of applications. This work contributes to the advancement of this goal, by producing a probabilistic modular NoC model for the verification of PSN properties. The architecture of this model has been verified, through formal properties which ensure that its behavior is in line with previous NoC attributes.

The modular model developed in this work was used to demonstrate its scalability. This resulted in the collection of PSN data under synthetic uniform distribution traffic patters, for both the entire model, and the individual routers. Results showed that the probability of PSN properties is not uniform, and is additionally biased by the implementing X-Y routing.

Future research includes implementing an NoC for simulating flit generation actually found in existing applications, as opposed to using the synthetic uniform distribution traffic pattern used in this and previous work. This would open the door for more customized results, tailored to specific applications. Research opportunities also exist for automating the comparative verification process. This would allow for a scripted process to track down the source of discrepancies between models swiftly and accurately, speeding up NoC verification. Additionally, the opportunity exists to create tools for automating model conversion into physical hardware designs. The ability to generate hardware (e.g. schematics or VHDL), from formal models would greatly improve NoC production, by reducing time spent in development, and mitigating human error.

REFERENCES

[1] J.-J. Lecler and G. Baillieu, “Application driven network-on-chip architecture exploration and refinement for a complex SoC,” *Design Autom. for Emb. Sys.*, vol. 15, pp. 133–158, 06 2011.

[2] Tsai, Lan, Hu, and Chen, “Networks on chips: Structure and design methodologies,” vol. 2012, 2012.

[3] P. Royannez, H. Mair, F. Dahan, M. Wagner, M. Streeter, L. Bouetel, J. Blasquez, H. Clasen, G. Semino, J. Dong, D. Scott, B. Pitts, C. Raibaut, and U. Ko, “90nm low leakage SoC design techniques for wireless applications,” in *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*, 2005, pp. 138–589 Vol. 1.

[4] N. K. R. Becchu, V. M. Harishchandra, and N. K. Yernad Balachandra, “System level fault-tolerance core mapping and FPGA-based verification of NoC,” *Microelectronics Journal*, vol. 70, pp. 16–26, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026269217302884

[5] S. Y. Jiang, G. Luo, Y. Liu, S. S. Jiang, and X. T. Li, “Fault-tolerant routing algorithm simulation and hardware verification of NoC,” *IEEE Transactions on Applied Superconductivity*, vol. 24, no. 5, pp. 1–5, 2014.

[6] J. Müller, M. R. Fadiheh, A. L. D. Antón, T. Eisenbarth, D. Stoffel, and W. Kunz, “A formal approach to confidentiality verification in SoCs at the register transfer level,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 991–996.

[7] R. Roberts, B. Lewis, A. Hartmanns, P. Basu, S. Roy, K. Chakraborty, and Z. Zhang, “Probabilistic verification for reliability of a two-by-two network-on-chip system,” in *Formal Methods for Industrial Critical Systems*, A. Lluch Lafuente and A. Mavridou, Eds. Cham: Springer International Publishing, 2021, pp. 232–248.

[8] B. Lewis, A. Hartmanns, P. Basu, R. Jayashankara Shridevi, K. Chakraborty, S. Roy, and Z. Zhang, “Probabilistic verification for reliable network-on-chip system design,” in *Formal Methods for Industrial Critical Systems*, K. G. Larsen and T. Willemse, Eds. Cham: Springer International Publishing, 2019, pp. 110–126.

[9] L. Taylor and Z. Zhang, “Scaling up livelock verification for network-on-chip routing algorithms,” in *Verification, Model Checking, and Abstract Interpretation*, B. Finkbeiner and T. Wies, Eds. Cham: Springer International Publishing, 2022, pp. 378–399.

[10] R. Salamat, M. Khayambashi, M. Ebrahimi, and N. Bagherzadeh, “A resilient routing algorithm with formal reliability analysis for partially connected 3D-NoCs,” *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3265–3279, 2016.

[11] J. Sepulveda, D. Aboul-Hassan, G. Sigl, B. Becker, and M. Sauer, "Towards the formal verification of security properties of a network-on-chip router," in *2018 IEEE 23rd European Test Symposium (ETS)*, 2018, pp. 1–6.

[12] L. Alhubail and N. Bagherzadeh, "Power and performance optimal NoC design for CPU-GPU architecture using formal models," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 634–637.

[13] M. Kim, T. Hiroyasu, M. Miki, and S. Watanabe, "SPEA2+: Improving the performance of the strength pareto evolutionary algorithm 2," in *Parallel Problem Solving from Nature - PPSN VIII*, X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 742–751.

[14] Z. Zhang, W. Serwe, J. Wu, T. Yoneda, H. Zheng, and C. Myers, "An improved fault-tolerant routing algorithm for a network-on-chip derived with formal analysis," *Science of Computer Programming*, vol. 118, pp. 24–39, 2016, formal Methods for Industrial Critical Systems (FMICS'2014). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642316000125

[15] C. Glass and L. Ni, "Fault-tolerant wormhole routing in meshes," in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993, pp. 240–249.

[16] N. Mediouni and S. Hasnaoui, "Phosphorus: An ultra low footprint and energy consumption 3D NoC architecture," in *2017 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC)*, 2017, pp. 129–133.

[17] J.-P. Katoen, "The probabilistic model checking landscape," in *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016, pp. 1–15.

[18] S. Hart, M. Sharir, and A. Pnueli, "Termination of probabilistic concurrent program," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 356–380, jul 1983. [Online]. Available: https://doi.org/10.1145/2166.357214

[19] A. Hartmanns and H. Hermanns, "The modest toolset: An integrated environment for quantitative modelling and verification," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 593–598. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_51

[20] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[21] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, “The probabilistic model checker Storm,” *Int. J. Softw. Tools Technol. Transf.*, vol. 24, no. 4, pp. 589–610, aug 2022. [Online]. Available: https://doi.org/10.1007/s10009-021-00633-z

[22] E. M. Hahn, Y. Li, S. Schewe, A. Turrini, and L. Zhang, “iscasMc: A web-based probabilistic model checker,” in *FM 2014: Formal Methods*, C. Jones, P. Pihlajasaari, and J. Sun, Eds. Cham: Springer International Publishing, 2014, pp. 312–317.

[23] S. Song, J. Sun, Y. Liu, and J. S. Dong, “A model checker for hierarchical probabilistic real-time systems,” in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 705–711.

[24] L. Wang and F. Cai, “Reliability analysis for flight control systems using probabilistic model checking,” in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2017, pp. 161–164.

[25] M. Lakin, L. Parker, D amd Cardelli, M. Kwiatkowska, and A. Phillips, “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.

[26] M. Kwiatkowska, G. Norman, and D. Parker, “Symmetry reduction for probabilistic model checking,” in *Computer Aided Verification*, T. Ball and R. B. Jones, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 234–248.

[27] Christel Baier and Edmund M. Clarke and Vasiliki Hartonas-Garmhausen and M. Kwiatkowska and Mark Ryan, “Symbolic model checking for probabilistic processes,” in *International Colloquium on Automata, Languages and Programming*, 1997.

[28] K. Fisler and M. Y. Vardi, “Bisimulation and model checking,” in *Conference on Correct Hardware Design and Verification Methods*, 1999.

[29] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang, “INFAMY: An infinite-state Markov model checker,” in *International Conference on Computer Aided Verification*, 2009.

[30] R. Roberts, T. Neupane, L. Buecherl, C. J. Myers, and Z. Zhang, “STAMINA 2.0: Improving scalability of infinite-state stochastic model checking,” in *Verification, Model Checking, and Abstract Interpretation*, B. Finkbeiner and T. Wies, Eds. Cham: Springer International Publishing, 2022, pp. 319–331.

[31] H. L. S. Younes and R. G. Simmons, “Probabilistic verification of discrete event systems using acceptance sampling,” in *International Conference on Computer Aided Verification*, 2002.

[32] A. Legay, A. Lukina, L.-M. Traonouez, J. Yang, S. A. Smolka, and R. Grosu, “Statistical model checking,” in *Computing and Software Science*, 2019.

[33] P. Zuliani, C. Baier, and E. M. Clarke, “Rare-event verification for stochastic hybrid systems,” in *Proceedings of the 15th ACM International Conference on*

Hybrid Systems: Computation and Control, ser. HSCC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 217–226. [Online]. Available: https://doi.org/10.1145/2185632.2185665

[34] C. E. Budde, P. R. D'Argenio, A. Hartmanns, and S. Sedwards, "A statistical model checker for nondeterminism and rare events," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 340–358.

[35] C. E. Budde, P. R. D'Argenio, and A. Hartmanns, "Automated compositional importance splitting," *Science of Computer Programming*, vol. 174, pp. 90–108, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642318301503

[36] M. Kwiatkowska, G. Norman, and D. Parker, *Stochastic Model Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 220–270. [Online]. Available: https://doi.org/10.1007/978-3-540-72522-0_6

[37] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015. [Online]. Available: https://doi.org/10.1007/s10009-014-0361-y

[38] P. Basu, R. J. Shridevi, K. Chakraborty, and S. Roy, "IcoNoClast: Tackling voltage noise in the NoC power supply through flow-control and routing algorithms," *IEEE Trans. VLSI Syst.*, vol. 25, no. 7, pp. 2035–2044, 2017.