MODULAR VERIFICATION FOR NETWORK-ON-CHIP DESIGNS USING PROBABILISTIC VERIFICATION AND ASSUME-GUARANTEE REASONING

by

Nicholas Waddoups

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	Honjie Wang, Ph.D. Committee Member

David F. Feldon, Ph.D.
Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2026

ABSTRACT

Modular Verification for Network-on-Chip Designs
Using Probabilistic Verification and
Assume-Guarantee Reasoning

by

Nicholas Waddoups, Master of Science
Utah State University, 2026

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

Network-on-Chip (NoC) architectures are the de-facto on-chip communication technology for large digital System-on-Chips (SoCs) and require high quality verification methods to provide reliability and safety guarantees. One major challenge in NoC design and implementation is power supply noise (PSN), where fluctations in network traffic cause potentially disastrous variations in power deliver across the NoC leading to further issues. An additional challenge is demonstrating functional correctness of NoC designs is critical to ensure correct and reliable chip function. This thesis presents a threefold approach to improving the correctness and reliability of NoC designs. The first contribution is functional verification of a probabilistic 2×2 NoC model using the Modest Toolset and computational tree logic (CTL) properties. Second, probabilistic verification of an *abstracted* NoC model using the Modest Toolset is presented. Finally, functional verification for arbitrary $n \times n$-sized NoC model using the Dafny verification-aware language is shown.

(169 pages)

PUBLIC ABSTRACT

Modular Verification for Network-on-Chip Designs
Using Probabilistic Verification and
Assume-Guarantee Reasoning
Nicholas Waddoups

To satisfy increasing demands for computer chip performance in personal computing, mobile devices, and commercial server computing, a modern computer chip is constructed with tens (or hundreds) of small individual computing modules. Each of these modules must communicate with one other to share information about the running state of a computer. Historically, when chips were a few modules a simple communication method sufficied. However, as the number of modules in a chip grew, a more effecient method was needed in order to maintain performance across the system as a whole. A Network-on-Chip (NoC) design is the de-facto communication method for chips with many components because it is effecient, performant, and scalable. NoC designs are more complex than historical designs and require in-depth testing, analysis, and verification to ensure that they function correctly. Not only do NoC designs need to function correctly, but they also must be rigorously analyzed in regards to their reliability. While testing provides a measure of confidence in a design, analysis of a system using mathematical methods, i.e., *formal verification*, provides strong guarantees that a system functions correctly and demonstrates specific reliability characteristics. This thesis first presents a formal analysis of the functional correctness of a mesh-style NoC design, and the provides a formal analysis of the reliability of that design in regard to electrical noise in the system. These contributions entail a NoC model that is highly flexible, scalable, and provably correct. This model will assist computer engineers in the creation of reliable and correct NoCs.

To Rachel

ACKNOWLEDGMENTS

I'd like to thank the many people in my life who have encouraged, supported, and pushed me to complete this research.

I'd be amiss to skip acknowledging my parents, who for many years encouraged my interest in math and science, supported my hobbies in robotics, and always encouraged me to seek knowledge and learning. My parents were the first to teach me to be critically minded, thoughtful in my communication, and diligent in my studies.

Additionally, I'd like to thank my advisor for sharing his enthusiasm in formal methods with me and inviting me to be critically minded. Dr. Zhang would commonly ask, "are you convinced?" in response to the presentation of a new idea or technique. This question has become one of my favorites to ask myself after learning something new.

Finally, I'd like to thank my incredible wife who has supported me through the past several years without fail. Rachel has always been there to support my academic pursuits, whether by encouraging my successes, talking through tough engineering problems with me, or by helping me get back up after failures.

And now, I'd like to acknowledge you dear reader, thank you for reading, please enjoy!

Nick Waddoups

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

#SAT Propositional Model Counting

ADT Algebraic Data Type

AG Assume-Guarantee

CPS Cyber-Physical System

CTL Computational Tree Logic

CTMC Continuous-Time Markov Chain

DTMC Discrete-Time Markov Chain

FIFO First-In First-Out

ITP Interactive Theorem Prover

MA Markov Automata

MCS Mixed-Criticality System

MDP Markov Decision Process

MPSoC Multi-Processor System-on-Chip

NoC Network-on-Chip

PCTL Probabilistic Computational Tree Logic

PE Processing Element

PMC Probabilistic Model Checking

PSN Power Supply Noise

PTA Probabilistic Timing Analysis

SHA Stochastic Hybrid Automata

SMC Statistical Model Checking

SPEA2 Strength Pareto Evolutionary Algorithm2

SSAT Stochastic boolean SATisfiability

TA Timed Automata

WCET Worst-Case Execution Time

LIST OF NOTATIONS

$\mathcal{C}$ The set of connections between routers in a Network-on-Chip

$\mathcal{N}$ A defined, mesh-style Network-on-Chip

p_d Packet destined for R_d

R A router

R_i Router i

$R_i.B_j$ Buffer j of R_i

$R_i.C_j$ Channel j of R_i

$\mathcal{R}$ The set of routers in a Network-on-Chip

CHAPTER 1

Introduction

Digital designs are continuously driven to deliver higher performance. Historically, gains were achieved through frequency and transistor scaling and single-core micro-architectural optimizations. However, as these methods hit physical limits, the industry shifted toward on-chip parallelism. Modern consumer-grade CPUs now integrate multiple processing cores and accelerators, relying on *Network-on-Chip* (NoC) technology for on-chip, inter-module communication. While NoCs enable efficient parallel communication, they introduce significant complexity not seen in previous bus or cross-bar communicaiton architectures. This work applies formal methods to establish high-confidence reliability and correctness guarantees for NoC designs *early* in the design life-cycle.

The reliability of NoC designs is critical, particularly for the complex *Multi-Processor System-on-Chips* (MPSoCs) now dominating the industry [1, 2]. As the communication backbone for processing cores, memory, and accelerators, a flawed NoC can render an entire chip partially or wholly inoperable. Beyond functional correctness, quantitative guarantees regarding performance and reliability are increasingly desirable for high-performance or safety-related NoCs [2–8].

Proving functional correctness demonstrates that a NoC implementation satisfies its specification [2,3], encompassing both *safety* and *liveness* properties. Key properties include ensuring packets eventually reach their destination (liveness), preventing buffer overflows (safety), and proving deadlock freedom (safety). Establishing these guarantees is a fundamental aspect of design verification.

A significant reliability challenge to NoC operation is PSN, a physically based phenomenon. Driven by fluctuating network traffic, PSN is caused by high current draw ($I(t)$) and sudden current shifts (di/dt) [4]. These current fluctuations cause timing violations that can corrupt data or lead to packet loss, thereby degrading system performance and re-

liability. Because PSN is heavily dependent on traffic patterns and router implementation, it requires robust analysis and characterization.

Correctness can be assessed via *formal methods*, *simulation*, or *emulation*. While simulation and emulation validate behavior over a limited set of execution traces, formal methods utilize mathematical techniques to prove properties over the entire state space of the system. This work leverages formal methods to derive guarantees for NoC behavior.

1.1 Contributions

This work advances the correctness and quantitative verification of NoCs with the following contributions:

1. **Optimization of the Modular NoC Framework:** An architectural improvement to the MODEST NoC framework originally presented in [7]. This optimization reduces the state space and model complexity, facilitating faster verification and improved scalability. The improvement provides additional modularity, allowing for more diverse verification.
2. **Extended PSN Analysis:** A comprehensive characterization of PSN on the MODEST NoC model using *Statistical Model Checking* (SMC). This includes analysis under the traffic patterns from [7, 8] for meshes up to 12×12, and the analysis of a *bursty* traffic pattern for 2×2, 3×3, and 4×4 topologies.
3. **Novel Router Activity Metric:** A novel metric that decouples PSN analysis from global NoC-wide counters. Unlike prior approaches that aggregate noise events into a single system-wide value, this metric tracks activity at individual routers. This granularity enables the identification of specific "hotspot" routers and correlates more precisely with physical layout-dependent noise phenomena.
4. **CTL-based Correctness Verification:** Formal verification of the MODEST NoC model using newly introduced CTL operators within the MODEST TOOLSET.
5. **Deductive Verification in Dafny:** The development and verification of a semantically similar NoC model in the Dafny language. This model uses *Assume-Guarantee* (AG) reasoning to verify the functional verification of the NoC model for arbitrary

NoC sizes. Additionally, it demonstrates the capabilities of Dafny for hardware verification and serves as a cross-verification reference for the MODEST model.

CHAPTER 2

Preliminaries

This chapter covers the theoretical foundations and technical context needed to understand the contributions presented in this thesis. The discussion covers many topics, beginning with an overview of NoCs and the architectural constraints relevant to this work. This chapter then introduces the formalisms required for probabilistic verification and deductive software verification. Finally, it provides technical language references for the specific tools employed: the MODEST modeling language and the Dafny programming language.

Due to the variety of topics covered, this chapter is structured primarily as a reference. Readers already familiar with specific domains–such as synchronous digital logic or the syntax of Dafny–may treat these sections as optional, consulting them only as needed to clarify notation or terminology used in later chapters. References back to this chapter are common and may help the unfamiliar reader navigate the contributions of this thesis.

2.1 Digital System Fundamentals

Digital hardware designs can be broadly categorized by their timing mechanisms as either an asynchronous and synchronous system. Asynchronous systems operate without a global clock signal, where state transitions are triggered by the completion of operations or changes in input signals. While both approaches can be successful for digital design, this work considers only synchronous digital design.

In synchronous digital design, all state transitions are coordinated via a global clock signal at a fixed frequency. This imposes a discrete notion of time, where the system evolves in lockstep at each timestep. For formal verification, this restriction is highly advantageous: it allows the real-world continuous physical behavior of a circuit to be abstracted into a sequence of discrete states. A synchronous digital system can therefore be modeled as a transition system, where the complexity of continuous time is removed.

2.1.1 Clock Cycles and State Updates

The fundamental unit of time in a synchronous system is the *clock cycle*. The system's operation relies on the separation of state-holding elements (sequential logic, such as flip-flops or registers) and data-processing elements (combinational logic).

Updates during each clock cycle t occur as follows:

1. Steady state at t: At the beginning of any clock cycle t, the sequential elements hold the current state, denoted as σ_t. These values are stable during cycle t and available to combinational logic blocks.

2. Update computation: During clock period t, the combinational logic evaluates the current state σ_t and any external inputs I_t. The combinational logic then computes the *next state* values.

3. Transitions from $t \to t+1$: On the active edge of the clock (typically the rising edge), the computed next-state values are captured by the sequential circuit elements. This instantaneous update transitions the system to state σ_{t+1}.

In the context of NoC verification, the behavior of the NoC is modeled as an initial state σ_0 and an update process δ that computes σ_{t+1} for any σ_t. This thesis presents two modeling approaches for this, one using the MODEST language and one using the Dafny language.

2.2 Network-on-Chip Fundamentals

The advent of multi-core chips in the 1990s spurred research of on-chip networks as a replacement for the traditional data bus [9]. Continually increasing core counts have necessitated the introduction of a low-latency, high-throughput communication solution. NoC architectures vary dramatically, but we will describe the relevant architectures here.

2.2.1 Network-on-Chip Components and Topology

A NoC is an on-chip network that connects different *Processing Elements* (PEs) together. The PEs may be CPUs, GPUs, acclerators, shared caches, or other I/O [9]. Each

PE is connected to a *router* in the NoC. The NoC is comprised of a interconnected network of routers that communicate with each other. As with traditional computer networks, NoCs can be arranged in a variety of unique topologies. The implementation of the router, topology of the NoC, and connection protocol to the PEs is dependent on the design goals of the project.

A standard NoC topology is the 2-dimensional square $n \times n$ mesh, as shown in Figure 2.1. This topology is used in 11 of the 16 real-world NoCs surveyed in [9] and [6–8]. The mesh network is made up of *router* nodes responsible for packet forwarding across the network via input *First-In First-Out* (FIFO) buffers and output channels. Each router connects to a local PE, such as a CPU, accelerator, or cache, which injects packets into the network. The router then arbitrates and forwards these packets toward their destinations.

Traffic, in the form of *packets*, is introduced to the NoC by PEs. Each packet contains a header and data. Typically, the header must contain the identifier of the destination router so that the packet can be routed across the network. New packets are added to the NoC by the PE connected to each buffer. The PE pushes new packets into the local buffer of a router, given that the local buffer isn't full, and then the router processes that packet and sends it toward its destination.

Definition 2.1. A packet p_i is defined as $\langle \mathcal{H}, \mathcal{D} \rangle$ where $\mathcal{H}$ is the header of the packet and $\mathcal{D}$ is the data in the packet. Both $\mathcal{H}$ and $\mathcal{D}$ must be binary-representable, as they are encoded as bits in the NoC. In our work, we only consider packets that meet the following constraints:

$$\begin{aligned} \mathcal{H} &= \langle id \rangle \ \text{(where the router } R_{id} \text{ is the packet's destination)} \\ \mathcal{D} &= \emptyset \end{aligned} \tag{2.1}$$

Remark. In other words, we only consider packets that store the identifier of the destination router. These constraints are imposed to reduce the state space and because in our abstracted NoC model we do not need the data in the packet.

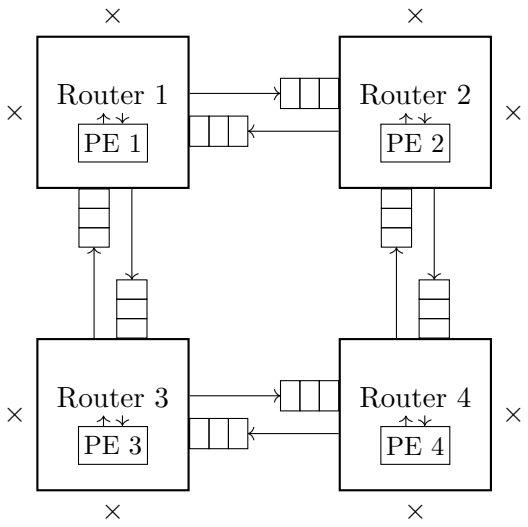


Fig. 2.1: 2x2 Mesh NoC Example

Routers contain fixed-size FIFO *buffers* to store incoming network packets and *channels* to send packets. In our mesh network, each router has five buffers and channels, defined by their location relative to the router: north (`N`), east (`E`), south (`S`), west (`W`), local (`L`). Figure 2.2 shows buffers as boxes (□□□) and channels as arrows (→). Buffer j in router i is referred to by $R_i.B_j$. The local buffer is used by the PEs to add new packets to the network. The four cardinal direction buffers are used to *receive* packets from neighboring routers. Channels are used to *send* packets to neighboring routers, or, when a packet has reached its destination, to the connected PE.

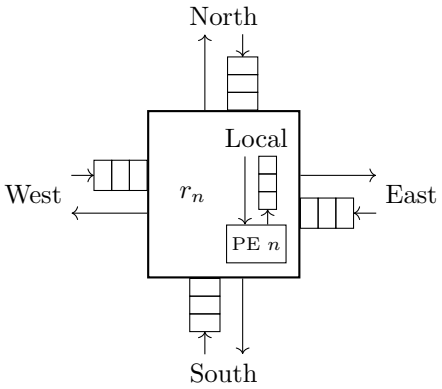


Fig. 2.2: Single Mesh NoC Router

In our NoC, buffers are synchronous FIFO buffers. These buffers typically have the following four capabilities: enqueue, dequeue, peek, and len. In a digital circuit, all these operations are synchronous and may occur once per clock edge. In our model, we allow for up to all four actions to happen on a single clock edge, meaning that a packet may be dequeued at the same time a new packet is enqueued.

Definition 2.2. A buffer $R_i.B_j$ is defined as a bounded list of packets $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ where p_0 represents the *head* of the list and p_{n-1} the *tail*. The maximum length of the list is n. Buffers have four associated operations.

- `enqueue`$(R_i.B_j, p) \to B$ is a function that returns a new list with p appended to the list as the new tail.
- `dequeue`$(R_i.B_j) \to B$ is a function that returns a new list with the front item dropped from the list.
- `peek`$(R_i.B_j) \to p$ is a function that returns the packet at the head of the list.
- `len`$(R_i.B_j) \to m$ is a function that returns the current number of elements in the buffer m.

Channels represent physical wires between different routers in the network. We consider the channel delay to be zero clock cycles in our NoC model, and we only allow channels to be used once per clock cycle. This is because in real digital circuits a wire should be assigned one value per clock cycle.

A router additionally contains an *arbiter* and *routing logic*. The arbiter determines which packets should be forwarded every clock cycle, and works with the routing logic to determine where each packet is forwarded to.

Definition 2.3. A router R_i is defined as $\langle \mathcal{B}, \mathcal{C}, \mathcal{A}, \mathcal{L} \rangle$ where $\mathcal{B}$ is a set of buffers, $\mathcal{C}$ is a set of channels, and $\mathcal{A}, \mathcal{L}$ are the arbiter and switching logic, respectively. For R_i

$$\mathcal{B} = \{R_i.B_b \mid b \in \{\mathtt{N}, \mathtt{E}, \mathtt{S}, \mathtt{W}, \mathtt{L}\}\}$$
$$\mathcal{C} = \{R_i.C_b \mid b \in \{\mathtt{N}, \mathtt{E}, \mathtt{S}, \mathtt{W}, \mathtt{L}\}\} \tag{2.2}$$

Figure 2.3 shows a conceptual router connected to a PE. This figure shows the five input buffers on the left, the five output channels on the right, and the aribter and routing logic as blocks internal to the router. The connection between the local channel, PE, and local buffer is made clear. From the prespective of the router, packets are sent to the PE via the local channel, and are received from the PE via the local buffer. Figure 2.2 shows a single router where each input buffer and output channel are graphically arranged as they would be in a mesh NoC, such as the NoC shown in Figure 2.4.

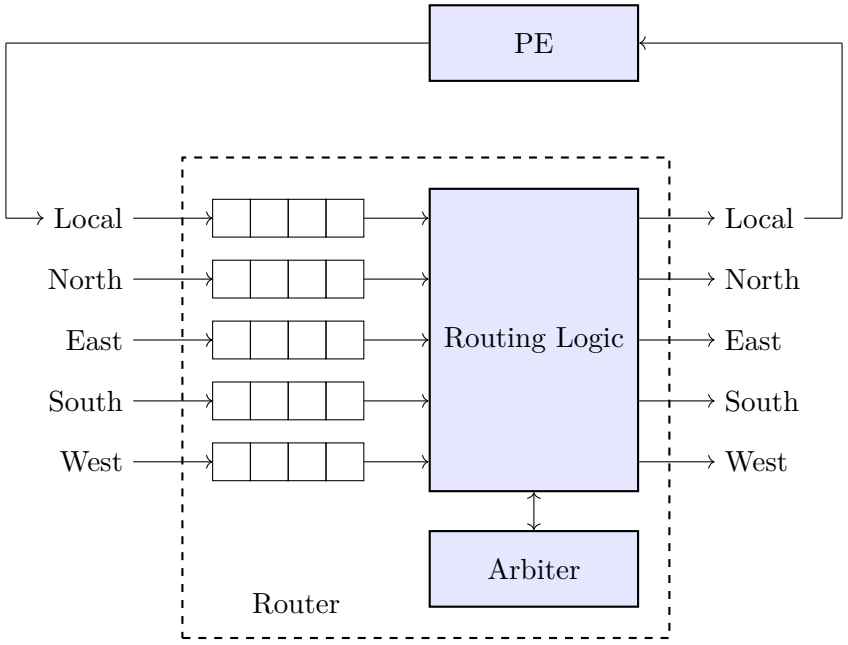


Fig. 2.3: Conceptual Router and Processing Element (PE) Diagram

A NoC in this work is a square mesh-style network consisting of n^2 routers where n

is the dimension of the router in $n \times n$. Routers are labeled with IDs ranging from 0 to $n^2 - 1$ starting at the top left of the network, and continuing left-to-right down each row of the network. See Figure 2.4 for a visual depiction of this. Each router is connected to it's neighbor with a buffer for inputs and a channel for outputs. Connections occur relative to the routers. For example, if R_0 is to left of R_1, then the connection between the two would be $R_0.C_{\mathtt{E}} \rightarrow R_1.B_{\mathtt{W}}$.

Definition 2.4. A NoC $\mathcal{N}$ is defined as the tuple $\langle \mathcal{R}, \mathcal{C}, n \rangle$ where $\mathcal{R}$ is the set of routers in $\mathcal{N}$, $\mathcal{C}$ is a set of connection pairs, and n is the $n \times n$ dimension of $\mathcal{N}$. For a valid mesh-style NoC, the following hold.

$$\mathcal{R} = \left\{ R_i \mid 0 \leq i < n^2 \right\} \tag{2.3}$$

$$\mathcal{C} = \{ (R_i, R_j) \mid R_i, R_j \in \mathcal{R} \wedge R_i \neq R_j \wedge (j = i \pm 1 \vee j = i \pm n) \} \tag{2.4}$$

The constraints on $\mathcal{C}$ enforce the mesh-style network, where a router can only have neighbors in each of the four cardinal directions.

As previously discussed, packets are introduced to the network by PEs. Packets move between neighboring routers by moving from a source buffer to a destination buffer via an intermediate channel. However, due to physical bandwidth constraints, the channel bit-width is often narrower than the packet size, meaning that packets can't be sent between routers in one clock cycle. To resolve this, hardware implementations serialize packets into smaller flow control units, or *flits*. For instance, a 64-byte packet transmitted over a 16-bit channel requires serialization into 32 flits, occupying the channel for 32 clock cycles.

A *single-flit packet* is assumed in this the NoC model used in this thesis. Under this assumption, the logical packet size aligns with the channel width, permitting atomic transfer between routers in a single clock cycle. This abstraction is motivated by three primary factors:

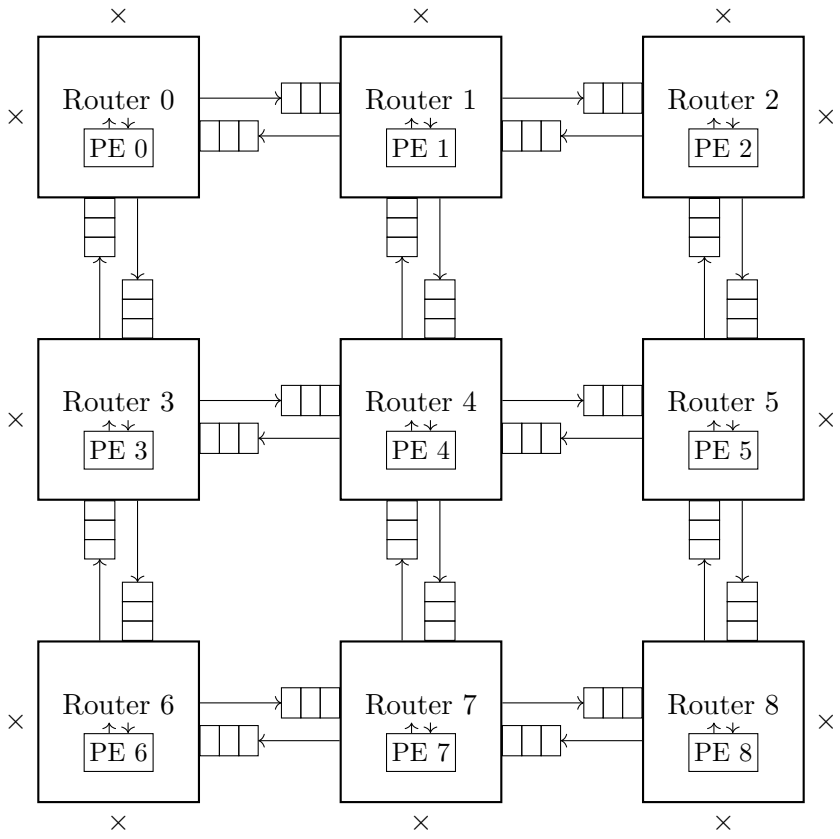


Fig. 2.4: 3×3 Mesh NoC Diagram

1. **Design Stage Agnosticism:** The PSN characterization targets the *early* architectural exploration phase, where specific parameters such as channel bit-width or total packet size may not yet be defined.
2. **Abstraction Level:** We prioritize the verification of macroscopic network properties–such as topological correctness, routing logic, and traffic patterns–over link-level circuit details. This abstraction enhances model scalability without compromising the validity of high-level performance metrics.
3. **Formal Tractability:** Multi-flit modeling (e.g., wormhole routing) requires tracking the state of head, body, and tail flits, which induces a state-space explosion (Sec. 2.6.1) that renders probabilistic model checking (via MODEST) intractable. While deductive verification (via Dafny) is theoretically less susceptible to this state explosion, we maintain the single-flit assumption across both methodologies (Chapter 6) to ensure

consistency and comparative validity.

Given this abstraction, the atomic unit of data transfer effectively represents both the packet and the flit; therefore, the terms are used interchangeably throughout this work.

The action of sending a packet from one router to another involves the sending router removing the packet from an internal *source* buffer, sending the packet over an output *channel*, and the receiving router adding the packet to the tail of the *destination* buffer. For packets that have reached their destination, they are removed from the source buffer, sent over the local channel, and received by the PE.

Definition 2.5. Let R_i, R_j, and R_x be routers in a NoC, where R_i and R_j are adjacent. $\sigma_k = (p_x, R_i.B_s \rightarrow R_i.C_c \rightarrow R_j.B_d)$ is the action σ_k of a packet destined for R_x from $R_i.B_s$ over channel $R_i.C_c$ to $R_j.B_d$. For this action to occur, buffer s and channel c cannot have been used yet on the current clock cycle, and buffer j must not be full. Semantically, the following steps occur for each buffer.

$$\begin{aligned} R_j.B_d{}' &= \texttt{enqueue}(R_j.B_d,\ \texttt{peek}(R_i.B_s)) \\ R_i.B_s{}' &= \texttt{dequeue}(R_i.B_s) \end{aligned} \tag{2.5}$$

2.2.2 Core Verification Challenges

Functional verification evaluates whether a NoC design adheres to its specification. It addresses critical correctness concerns, such as "will the NoC reach an illegal state?" or "does the design permit data corruption?" We formulate these concerns as formal *properties* to be checked against a model of the NoC. The objective is to derive a proof demonstrating that these properties *hold* (remain true) across all reachable states and execution paths of the model.

NoC verification broadly encompasses the validation of network logic and behavior. This thesis distinguishes between two verification categories: (1) *functional verification*, which targets logical correctness properties including safety (e.g., deadlock freedom) and

liveness (e.g., guaranteed packet delivery); and (2) *probabilistic verification*, which addresses quantitative metrics such as reliability and performance. For functional verification, we focus specifically on formal methods that demonstrate proof of system correctness, while for probabilistic verification, we use statistical methods to verify reliability.

AG–also termed *compositional* or *modular* reasoning–provides a scalable framework for verifying complex systems. AG reasoning decomposes the system into modular components, verifies them in isolation, and subsequently composes the component-level proofs to establish system-wide correctness ([10, Ch. 12]). Floyd and Hoare introduced an early AG framework for software in the late 1960s, assigning a *precondition* (assumption) and *postcondition* (guarantee) to every program statement [11,12]. In *Hoare logic*, program correctness follows if the postcondition of a statement implies the precondition of the subsequent statement. Lamport expands this to general compositional methods, delineating the specific conditions under which AG reasoning proves most effective [13].

2.3 Power Supply Noise Abstraction

PSN in NoC architectures comes primarily from two components: *resistive noise*, governed by the product of circuit resistance and instantaneous current (IR), and *inductive noise*, driven by the rate of current change through parasitic inductance ($L\frac{di}{dt}$). These voltage fluctuations can severely degrade signal integrity, potentially causing *timing errors* that compromise network reliability. A timing error occurs when a circuit latency exceeds the clock period, resulting in data corruption, loss, or spurious signals. While circuit-level analysis of PSN offers precision, it is computationally intractable for formal verification and unavailable during early-stage architectural exploration. Consequently, previous work [6] adopted a behavioral abstraction that correlates router activity with physical noise phenomena. This correlation was shown by Basu et al. [4] and utilized in prior formal NoC characterizations by [6–8].

2.3.1 Behavioral Modeling

In the behavioral model proposed by [6], current consumption is approximated by

router activity, defined as the number of packets processed by a router in a single clock cycle. For a standard 5-port router, this activity α ranges from 0 (idle) to 5 (saturation). Following the correlation established in [4], high activity during a single clock cycle maps to resistive noise and abrupt activity fluctuations between clock cycles maps to inductive noise.

Definition 2.6. Activity Trackers.

To formally track these events globally in a NoC over a sequence of clock cycles, [6] introduces two cumulative counters, $\mathcal{C}_{res}$ and $\mathcal{C}_{ind}$, which operate based on user-defined thresholds (T_{res} and T_{ind}). Additionally, the activity α of each router is tracked per each clock cycle t, denoted as α_t.

- **Resistive noise**: $\mathcal{C}_{res}$ increments if the current activity α_t exceeds or equals T_{res}.
- **Inductive noise**: $\mathcal{C}_{ind}$ increments if the absolute change in activity $|\alpha_t - \alpha_{t-1}|$ exceeds or equals T_{ind}.

2.3.2 Formal Specification of Power Supply Noise

PCTL formulas and *Probabilistic Model Checking* (PMC) *upper-bound* the probability of PSN noise events for each clock cycle. As detailed in Section 2.5, we utilize the bounded eventually operator to verify these properties up to N clock cycles. Here, time progression is modeled via a reward annotation, accumulate(clk), which represents the sum of elapsed clock cycles for an execution path.

Equation 2.6 defines the property checking the probability that the resistive noise count exceeds a limit K within N cycles. Equation 2.7 defines the analogous property for inductive noise.

$$p_r = P_{=?}[\Diamond^{[\text{accumulate}(clk)\leq N]}\, \mathcal{C}_{res} \geqslant K] \tag{2.6}$$

$$p_i = P_{=?}[\Diamond^{[\text{accumulate}(clk)\leq N]}\, \mathcal{C}_{ind} \geqslant K] \tag{2.7}$$

Equations 2.6 and 2.7 establish an upper bound on the probability of exceeding or

equalling K PSN events by the semantics of the bounded eventually operator. This operator ($\Diamond$) aggregates the probability mass of all execution traces where a counter $\mathcal{C}$ is greater than or equal to K at *any* point within $[0, N]$ clock cycles. Therefore, the result captures the cumulative probability that a PSN event occurs, rather than the instantaneous probability at a specific clock cycle. For instance, even if the probability of a violation at a single instant is low, the probability of a violation occurring *eventually* during a prolonged window may be significant; this operator accounts for that cumulative risk.

2.4 Formal Models for System Verification

DTMCs and *Markov Decision Processs* (MDPs) are mathematical models used to verify probabilistic properties of systems. This section gives an overview and example of each modeling system.

2.4.1 Discrete Time Markov Chains

DTMCs are a stochastic model for systems that exhibit probabilistic behavior and transition between states in discrete time steps. The PSN analysis in this work is accomplished by underlying DTMCs.

Definition 2.7. A labelled DTMC with rewards is a tuple $D = \langle S, \bar{s}, P, L, AP, \underline{\rho}, \iota \rangle$ where:

- S is a finite set of *states*.
- $\bar{s} \in S$ is the *initial state*.
- $P : S \times S \to [0, 1]$ is the *transition probability matrix*, satisfying $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$.
- AP is a finite set of *atomic propositions*.
- $L : S \to 2^{AP}$ is a *labeling function* assigning a set of atomic propositions to each state.
- $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$ is a *state reward function*.
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ is a *transition reward function*.

The matrix P describes the probabilistic system dynamics, where $P(s, s')$ represents the probability of transitioning from state s to s'. The labeling function L maps states to properties of interest (e.g., $error \in L(s)$), facilitating logic-based verification. Rewards, defined by $\underline{\rho}$ and ι, allow for quantitative analysis; $\underline{\rho}(s)$ accumulates value over steps spent in a state, while $\iota(s, s')$ assigns a cost or reward to a transition.

The execution of a DTMC is characterized by *paths*.

Definition 2.8. A path ω in a DTMC D is a non-empty, potentially infinite sequence of states $\langle s_0, s_1, s_2, \ldots \rangle$ such that $s_0 = \bar{s}$ and for all $i \geq 0$, $P(s_i, s_{i+1}) > 0$. We denote the i-th state of path ω as $\omega(i)$ and the length of a finite path as $|\omega|$.

Example: Server Availability

We illustrate DTMCs with a simple server model, D_{server}, depicted in Figure 2.5. The system consists of three states: $S = \{s_0, s_1, s_2\}$, representing *Idle*, *Busy*, and *Down* respectively. The propositions are defined as $AP = \{idle, busy, down\}$.

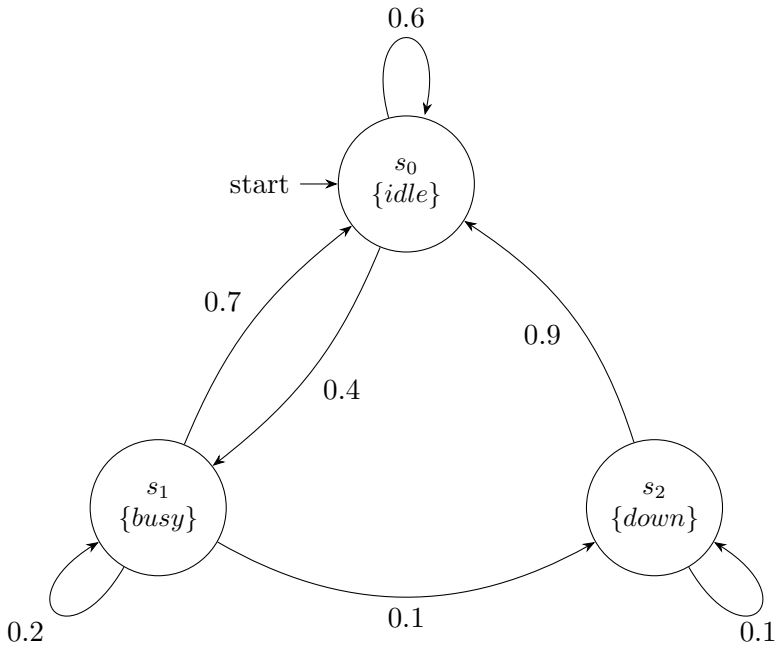


Fig. 2.5: DTMC for D_{server}

The probabilistic system is defined as follows: the server accepts jobs with probability 0.4 ($s_0 \to s_1$). Once busy, it completes a job ($s_1 \to s_0$, $p = 0.7$), continues processing ($s_1 \to s_1$, $p = 0.2$), or crashes ($s_1 \to s_2$, $p = 0.1$). A crashed server reboots with probability 0.9.

We assign a state reward to s_2 to penalize downtime ($\underline{\rho}(s_2) = 1$) and transition rewards to quantify energy consumption during job processing ($\iota(s_0, s_1) = 1, \iota(s_1, s_1) = 1$). The formal definition is given in Equation 2.8.

$$
\begin{aligned}
P &= \begin{bmatrix} 0.6 & 0.4 & 0 \\ 0.7 & 0.2 & 0.1 \\ 0.9 & 0 & 0.1 \end{bmatrix} & L(s) &= \begin{cases} \{idle\} & \text{if } s = s_0 \\ \{busy\} & \text{if } s = s_1 \\ \{down\} & \text{if } s = s_2 \end{cases} \\
\underline{\rho}(s) &= \begin{cases} 1 & \text{if } s = s_2 \\ 0 & \text{otherwise} \end{cases} & \iota(s, s') &= \begin{cases} 1 & \text{if } (s, s') \in \{(s_0, s_1), (s_1, s_1)\} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
\tag{2.8}
$$

2.5 Temporal Logics for Specification

Boolean logic alone is insufficient for specifying temporal behaviors, instead we use temporal logics that extend Boolean logic with temporal operators.

2.5.1 Safety and Liveness Properties

A *safety* property asserts that a specific invariant is maintained. Intuitively, it encodes the idea that "nothing bad ever happens". A *liveness* property asserts that a desirable state is eventually reached. Intuitively, it encodes the idea that "something good will happen".

2.5.2 Computation Tree Logic

CTL is a *branching-time temporal logic* interpreted over the computation tree representing all possible executions from a given state [14, Ch. 6].

CTL uses *state formulae* (evaluated on states) and *path formulae* (evaluated on execution paths) to specify behavior. Path formulae utilize quantifiers $\exists$ ("there exists a path")

and $\forall$ ("for all paths"), combined with temporal operators: $\bigcirc$ (Next) and $\mathcal{U}$ (Until). These operators and the derived operators $\Diamond$ (Finally/Eventually) and $\Box$ (Globally/Always) are defined in Table 2.1.

Operator	**Mnemonic**	**Meaning**
$\exists\bigcirc\Phi$	EX	Exists a path where Φ holds in the ne**X**t state.
$\forall\bigcirc\Phi$	AX	For **A**ll paths, Φ holds in the ne**X**t state.
$\exists\Diamond\Phi$	EF	**E**xists a path where **F**inally Φ holds.
$\forall\Diamond\Phi$	AF	For **A**ll paths, **F**inally Φ holds.
$\exists\Box\Phi$	EG	**E**xists a path where **G**lobally Φ holds.
$\forall\Box\Phi$	AG	For **A**ll paths, **G**lobally Φ holds.
$\exists(\Phi\mathcal{U}\Psi)$	EU	**E**xists a path where Φ holds **U**ntil Ψ.
$\forall(\Phi\mathcal{U}\Psi)$	AU	For **A**ll paths, Φ holds **U**ntil Ψ.
		(Φ and Ψ are Boolean state formulae.)

Table 2.1: CTL Operators

2.5.3 Probabilistic Computation Tree Logic

PCTL extends CTL for DTMCs. Rather than asserting if a property is possible or guaranteed, PCTL quantifies the probability of a property.

In PCTL, the standard path quantifiers ($\exists, \forall$) are replaced by the probabilistic operator $P_{\sim p}[\phi]$, where $\sim \in \{<, \leq, \geq, >\}$ and $p \in [0, 1]$. This asserts that the probability measure of paths satisfying ϕ meets the bound $\sim p$. Additionally, if the probability measure is of interest, the $P_{=?}$ operator returns the probability of ϕ occurring. The logic supports the temporal operators ($\bigcirc$,$\mathcal{U}$) and their bounded variants ($\mathcal{U}^{\leq k}$), as shown in Table 2.2.

Path Formula ϕ	**Meaning (within $P_{\sim p}[\phi]$)**
$\bigcirc\Phi$	Φ holds in the next state.
$\Phi\mathcal{U}^{\leq k}\Psi$	Ψ holds within k steps; Φ holds until then.
$\Phi\mathcal{U}\Psi$	Ψ eventually holds; Φ holds until then.
$\Diamond\Phi$	Φ eventually holds.
$\Box\Phi$	Φ holds in every state.
$\Diamond^{\leq k}\Phi$	Φ holds within k steps.
$\Box^{\leq k}\Phi$	Φ holds for the first k steps.

Table 2.2: PCTL Path Formulae

This thesis employs *reward-bounded eventually* properties to characterize PSN. Equation 2.9 demonstrates this, where the accumulation of a transition reward ι is constrained by an upper bound k.

$$P_{=?}\left[\Diamond^{\sum \iota \leq k}\Phi\right] \tag{2.9}$$

Intuitively, this expression calculates the probability that Φ becomes satisfied before the accumulated reward ι exceeds k. In the context of the NoC model, this formulation captures properties such as "the probability that the number of inductive PSN events exceeds 5 within 10 clock cycles."

2.6 Formal Verification Techniques

This section outlines relevant research regarding formal verification methods. Particular attention is paid to available tools, as the research in this thesis is not to develop new tools, but rather to use existing tools.

2.6.1 Model Checking

Model checking is a highly automated method of formal verification that explores all possible states of a system to check if specified properties hold (Chapter 1 of [10]). Typically, model checking tools consist of a *model checker*, a *modeling language*, and a *property specification language*. The modeling language is used to describe the model and the properties to check, and then the model checker generates all the possible states of the system and checks the properties on them.

Figure 2.6 shows a high-level overview of model checking. To perform model checking on a system, the system is transformed into a finite-state model and the system requirements into a logic specification. Both are fed into a model checker, which verifies that the finite state model satisfies the logic specification. The model checker returns a result stating that the properties were satisfied (✓) or that they were not (×). If the properties were not satisfied a counterexample is given to show how the model failed the properties.

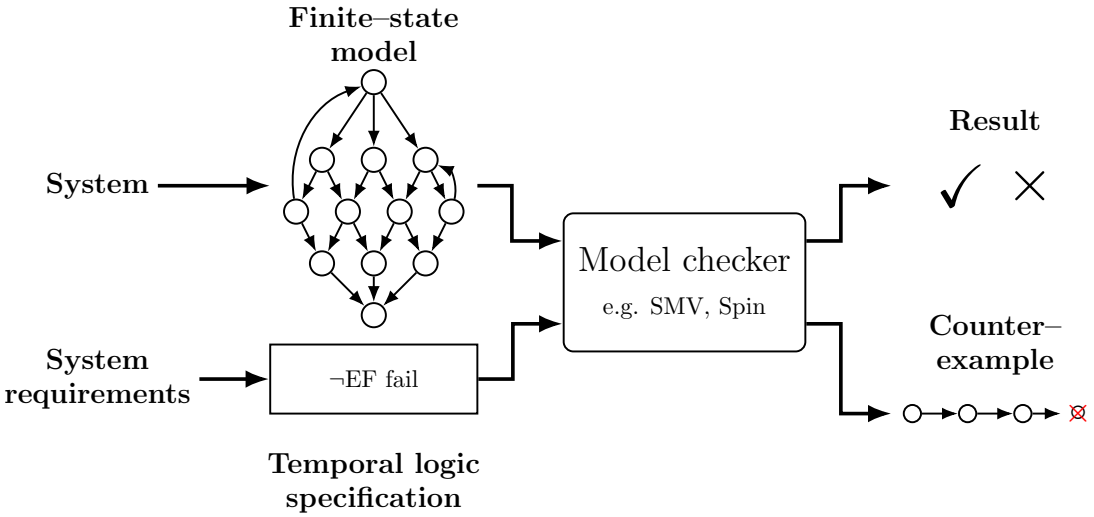


Fig. 2.6: High-Level Overview of Model Checking.

The SPIN model checker is designed for checking LTL properties on systems specified in the Promela modeling language [15]. The TLC model checker verifies systems and properties, both of which are specified in the TLA$^+$ language [16, 17]. CBMC [18] and ESBMC [19] are *bounded model checkers* (a special subset of model checking) that verify systems specified in the C programming language. NuSMV is a finite-state model checker that can check both synchronous and asynchronous systems and employs *symbolic model checking* to help reduce the number of states calculated [20]. nuXmv [21] extends NuSMV and allows for synchronous infinite-state systems to be described. UPPAAL is a toolkit that allows for real-time systems to be defined visually using a GUI and model checked [22].

One of the main challenges in all model checking applications is *state-space explosion*. Since model checking aims to enumerate the states of systems, consequently the verification efforts are bounded by the available memory of the system running the model checker. To combat these issues, successful methods were developed to abstract the state representation of systems. Key methods to combat this problem include symbolic state representation, predicate abstraction, binary decision diagrams, SAT/SMT-based model checking, interpolation, and AG reasoning [10]. The last of these, AG reasoning, is a central compositional technique explored in this thesis.

2.6.2 Probabilistic Model Checking

Probabilistic verification can answer questions such as, "will this bad event happen less than 5% of the time?", or "what is the probability that a message reaches its destination?" Like functional verification (Section 3.1.1) these questions are formulated into properties, and then checked against the formal model of the system. After the formal probabilistic verification process we will have quantitative guarantees about the behavior of the system. These guarantees can be used to qualify the reliability, performance, and probabilistic behavior of a system. Modern PMC tools include MODEST [23], PRISM [24], and STORM [25]. These tools generally operate on models encoded as Markov chains. PMC checkers suffer from the state explosion problem, just like their non-probabilistic model checking counterparts. Methods to reduce the state-space explosion problem in PMC are inherently more complex, due to the additional constraint of maintaining valid probabilities when shrinking the state space. Methods such as bisimulation reduction [26] and predicate abstraction [27] help reduce state explosion, but cannot remove the problem.

2.6.3 Statistical Model Checking

Statistical model checking (SMC) is a PMC method where probabilities are not precisely calculated, but rather statistically estimated over a sample of the state space [28]. The state space is sampled by simulating traces of the system. These traces are much less memory intensive compared to state space exploration, as they constitute only a single possible execution path through the model. Typically they are fast to generate, and multiple traces can be generated in parallel on multi-core machines. Statistical model checking does not provide the hard guarantees that PMC provides, but rather a quantitative analysis with statistical error bounds and confidence level. Methods such as rare-event simulation [29] help increase the confidence level that edge cases are covered.

2.6.4 Deductive Verification

Interactive theorem provers (ITP), or *proof assistants*, are programs that attempt to assist a user in the process of developing a formal proof, automating many lower-level logical

steps. Newell and Simon presented the theory and implementation of a basic theorem prover in 1956 [30]. Modern ITPs such as Rocq (formally Coq) [31], Lean [32], Isabelle/HOL [33], and ACL2 [34] provide strong capabilities for scalable verification of hardware and software systems using mathematical reasoning. While these tools are exceptionally powerful, they require significant user expertise in formal logic. This high barrier to entry motivates the development of verification-aware programming languages, such as Dafny (Section 3.3), which integrate verification capabilities directly into a more traditional software development workflow.

2.6.5 Floyd Logic and Hoare Triples

One popular method for verifying programs came from Robert Floyd in his 1993 paper, "Assigning Meaning to Programs" [12]. In this paper Floyd walks through a logical, flow chart-based method to determine the correctness of sequential programs. Often, Floyd logic is expressed through *Hoare triples*, a notation developed by C.A.R. Hoare that expresses Floyd Logic as mathematical statements, rather than flow charts [11]. The basis of Hoard logic is the Hoare triple, which is a three element tuple made up of a *precondition* P, a *statement* S, and a *post-condition* Q.

Definition 2.9. A Hoare triple $\{P\}S\{Q\}$ holds if $S \mid P \implies Q$ holds, i.e. the execution of statement S given the formula P implies the formula Q.

To reason about a program, one builds up a series of pre- and post-conditions and determines if certain properties are met. For example, consider the following program in Code Snippet 2.1 where we build up a series of post-conditions given a statement and a precondition. This example has three Hoare triples

$\{true\}$	`var x := 0`	$\{x = 0\}$
$\{x = 0\}$	`var y := x + 1`	$\{x = 0 \land y = x + 1\}$
$\{x = 0 \land y = x + 1\}$	`assert y > 0`	$\{(x = 0 \land y = x + 1) \implies y > 0\}$

We could also consider this entire example as one large Hoare triple

$$\{true\}\ \texttt{var x := 0, var y := x + 1, assert y > 0}\ \{(x = 0 \land y = x + 1) \implies y > 0\}$$

In this example we reason forward, starting first with the most general (weakest) precondition, *true*, and then generating the most restrictive (strongest) post-condition after every statement. If you are given a post-condition, it's also possible to work backward and generate the pre-condition for any statement.

The goal of Floyd logic and Hoare triples are to verify properties about a program. Typically, we would phrase this as saying that we want to *guarantee* that our program *does something* given some *assumption*. For example, in Code Segment 2.1 we want to check at the end of program that $y > 0$. In our toy example, we'll use `assert` to do this. This will trigger a check that has the form

$$\{P\}\ \texttt{assert(}Q\texttt{)}\ \{P \implies Q\}$$

If this check holds, then the assert is true, otherwise the program is invalid and should crash. Looking at our example program in Code Snippet 2.1 we can confidently state that the assert holds, which means that we have proved this property on our program. For a more in depth introduction to Floyd logic and Hoare triples, we refer the interested reader to "Program Proofs" by K.R. Leino [35].

Code Segment 2.1. Floyd Logic and Hoare Triples in Action

$\{true\}$
`var x := 0;`
$\{x = 0\}$
`var y := x + 1;`
$\{x = 0 \land y = x + 1\}$
`assert y > 0;`
$\{(x = 0 \land y = x + 1) \implies y > 0\}$

2.7 Verification Tools and Languages

2.7.1 The Modest Toolset

The MODEST TOOLSET [23] is a comprehensive verification environment capable of analyzing *Stochastic Hybrid Automata* (SHA) and *Markov Automatas* (MAs). These models, SHA and MAs, are a superset of DTMCs ans *Continuous-Time Markov Chains* (CTMCs), allowing MODEST to represent many different types of systems. This section outlines the relevant features of the MODEST language used in this thesis, as defined in the language reference.

The Modest Language

The MODEST language is a process-algebraic modeling language wherein every model is a process consisting of declarations and behavior. It supports a variety of formalisms, but this work focuses on its DTMC capabilities (a subset of MA).

Variables and Types. Modest supports standard primitive types including `bool`, `int`, and `real`. While integers are mathematically defined as values in $\mathbb{Z}$ (unbounded), they are usually represented as 32-bit integers. Bounded integers are declared using `int(min..max)`.

Complex data structures are supported through **arrays**, **option types**, and user-defined **datatypes** (structs):

- **Arrays:** Declared as `type[]`, representing sequences of values.
- **Options:** Declared as `type option`, which hold either a value of `type` or the empty value `none`.
- **Datatypes:** User-defined records declared with named members.

A major reason for choosing MODEST over other probabilistic modeling tools such as PRISM [24] is MODEST's robust support for complex datatypes. Without these data structures, scalable verification would not be tractable.

Variables may be declared as `const` (constant expressions) or `transient` (values not stored in the state space, they are only considered to exist during transitions) (Code Segment 2.2).

Code Segment 2.2. Variables and Types in MODEST.

```
int x;            // Unbounded integer
int(0..3) b;      // Integer bounded to {0,1,2,3}
const int N = 10; // Constant
transient int rew; // Transient variable (e.g., for rewards)
bool[] flags;     // Array of booleans
datatype Point = {int x, int y}; // User-defined datatype
```

Actions and Assignments. State updates and variable assignment occur in `{= ... =}` blocks. Typically, assignments within a block are executed simultaneously; the expressions on the right-hand side are evaluated based on the current state before any variables are updated. However, updates can be ordered with labels `1:` before each assignment in a `{=}` block. Assignment blocks with inconsistent updates, such as assigning two different values to a variable, are considered bad models in MODEST. Based on experimentation, MODEST may or may not catch these errors at compile time, leading to unexpected results during verification time. Because of this, it's best to ensure that updates are consistent, and that variables assignments do not occur simultaneously.

Assignments are coupled with **actions**, which facilitate synchronization. If no action is specified, the silent action τ (`tau`) is implied. Actions (Code Segment 2.3) are declared explicitly using the `action` keyword. Synchronization is useful when considering models that have parallel operations.

Code Segment 2.3. Actions and Assignments in MODEST

```
action sync;       // Declaration of an action
tau {= x = x + 1 =}; // Silent transition, no synchronization
sync {= x = 0, y = 1 =}; // Synchronizing transition
tau {= 1: x = y + 1,
      2: y = 0  =} // Ordered assignments
```

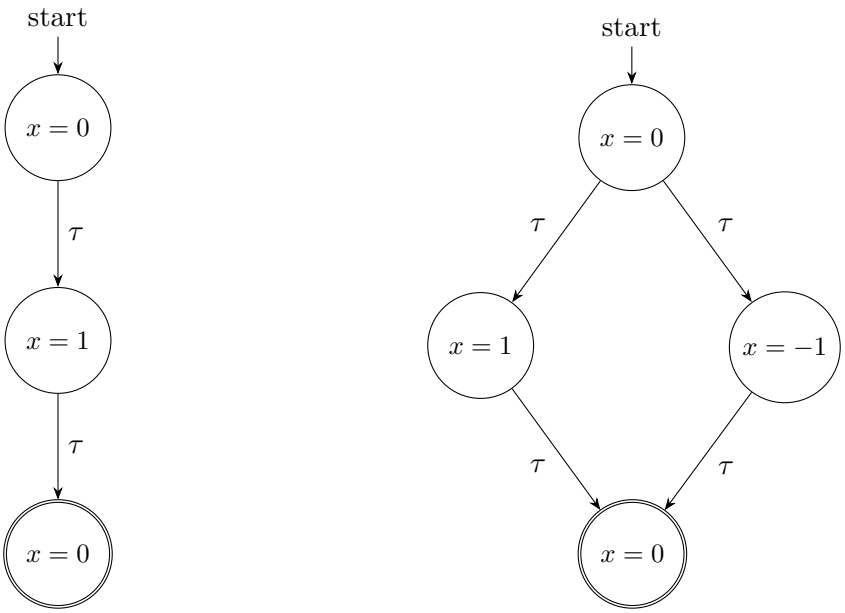
Composition and Control Flow. Behaviors are composed sequentially or in parallel (Code Segment 2.4). Sequential composition (`P; Q`) executes P then Q. Parallel composition is achieved via the `par` construct, which explores the interleaving of concurrent processes. Non-deterministic choice is expressed separately using the `alt` construct.

Figure 2.7 illustrates the difference between sequential execution and the interleaved state space generated by `par`.

Code Segment 2.4. Composition in MODEST

```
// Sequential: x becomes 1, then 0
{= x = x + 1 =}; {= x = x - 1 =}

// Parallel: Interleaved execution
par {
  :: {= x = x + 1 =}
  :: {= x = x - 1 =}
}
```



(a) Sequential Composition ($P; Q$) (b) Parallel Composition (`par`)

Fig. 2.7: Sequential vs. Parallel Composition in MODEST

Processes and Functions. Reusable behavior is encapsulated in `process` definitions, which may accept parameters. Pure computation is defined using `function` declarations, which map inputs to outputs without side effects.

Code Segment 2.5. Processes and Functions

```
// Process definition
process Worker(int id) { ... }

// Function definition
function int square(int n) = n * n;
```

Properties and Rewards

MODEST specifies properties using `property` declarations in the outermost scope. It supports both CTL and PCTL operators, including:

- **CTL:** `A[]` (for **A**ll paths globally ($\Box$)) and `E<>` (**E**xists eventually ($\Diamond$)).
- **PCTL:** `Pmax` / `Pmin` (probabilistic bounds) and `<>` (eventually).
- **Rewards:** Accumulated rewards are expressed via `S(e)`.

Code Segment 2.6 demonstrates a reward-bounded property on a DTMC. A `transient` variable `clk` sets the reward of a clock cycle transition to one. The property calculates the probability[1] that `C_res > 5` within a cumulative clock reward of 10 (within 10 clock cycles).

Code Segment 2.6. Reward Bounded Property

```
action nextClock;
transient int(0..1) clk; // Transient reward variable
int(0..) C_res = 0;

process Clock() { nextClock {= clk = 1 =} }

// ... model definition ...

// Probability that C_res > 5 within 10 clock cycles
property PSN_10 = Pmin(<> [S(clk) ⩽ 10] (C_res > 5));
```

2.7.2 The Dafny Language

Dafny [36] is a verification-aware programming language that combines functional specifications with imperative code. Unlike standard programming languages where verification

[1] Since DTMCs have no non-determinism $Pmin_{=?}(\Phi) = Pmax_{=?}(\Phi)$.

is an external process, Dafny includes verification constructs–such as preconditions, post-conditions, loop invariants, and assertions (`requires`, `ensures`, `invariant`, `assert`) and quantifiers (`forall`, `exists`)–directly in the syntax. The compiler generates a program representation for an SMT solver, which checks that the program satisfies its specifications. This section outlines the semantic features of Dafny uses in this thesis. For a comprehensive Dafny introduction and reference, see [35].

Variables and Assignment

Dafny uses a strong static type system with type inference. The language supports normal (deterministic) assignment (`:=`) and *non-deterministic assignment* with the *such that* operator (`:|`). This operator assigns a value chosen arbitrarily from the set of values satisfying a given predicate.

Code Segment 2.7. Variables

```
// Explicit type declaration
var x: int;

// Type inference with deterministic assignment
var y := 10;

// Non-deterministic assignment: z becomes an arbitrary value satisfying
   ↪ the predicate
var z :| 0 ⩽ z < 10;
```

Functions vs. Methods

Dafny distinguishes between mathematical functions and imperative methods (Code Segment 2.8). *Functions* are pure, side-effect-free mappings. They are evaluated atomically and must be deterministic. *Methods* represent imperative procedures that may modify state (heap), contain non-determinism, and generally correspond to the compiled code. Both constructs support AG reasoning via preconditions (assumptions) and postconditions (guarantees).

Code Segment 2.8. Functions and Methods

```
// Pure function: Used for specification, no side effects allowed
function add_one(n: int): int
   ensures add_one(n) > n
{
   n + 1
}

// Imperative method: Can contain non-determinism and modify state
method random(lower: int, upper: int) returns (r: int)
   requires lower < upper
   ensures lower ⩽ r ⩽ upper
{
   // Body satisfies the postcondition non-deterministically
   r :| lower ⩽ r ⩽ upper;
}
```

Classes and References

To model stateful systems, Dafny uses *classes*, which use reference semantics and a mutable heap (Code Segment 2.9). A class instance is a reference to an object containing fields. Unlike value types (like integers), using classes requires manual management of the *heap frame*–the memory locations a method is permitted to read or modify. The heap frame is managed with the `modifies` clause, which explicitly scopes side effects of a method.

Code Segment 2.9. Classes

```
class A {
   var x: int
   var a: array<int>

   constructor(x: int)
      requires x > 0
      ensures this.x == x && this.a.Length == x
   {
      this.x := x;
      this.a := new int[x];
   }

   method doThis(y: int) returns (z: int)
      modifies this.a // Explicitly claims write access to array 'a'
      // ...
}
```

Subset Types and Witnesses

Dafny allows the definition of *subset types*, which refine a base type with a predicate constraint (Code Segment2.10). Typically, Dafny requires proof of *type inhabitation*, i.e., that there exists at least one value satisfying the constraint. This proof is provided via a `witness`. If the compiler cannot automatically deduce a witness (e.g., 0 for integers), the user must provide one explicitly. If a type is possibly empty, the syntax `witness *` allows for treating the type as potentially uninhabited, restricting how it can be used in executable contexts.

Code Segment 2.10. Subset Types

```
// Positive integers: Witness '1' proves the type is inhabited
type Positive = i: int | i > 0 witness 1

// Even integers: Dafny automatically deduces '0' as a valid witness
type Even = i: int | i % 2 == 0

// A generic set of three elements: Inhabitation depends on T, so
   ↪ witness is unknown )type Three<T> = s: set<T> | |s| == 3 witness
   *
```

Algebraic Datatypes

Inductive algebraic datatypes (Code Segment 2.11) are immutable structures used to model groups of data (immutable structs). Unlike classes, they are value types. Dafny automatically generates induction principles for recursive datatypes, facilitating inductive proofs over structures like lists or trees. Pattern matching (`match`) is used to deconstruct these types and is exhaustively checked by the verifier.

Code Segment 2.11. Datatypes

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)

function Length<T>(l: List<T>): int {
   match l
   case Nil ⇒ 0
   case Cons(_, tail) ⇒ 1 + Length(tail)
}
```

Framing: Reads Clauses

While functions are side-effect free, they may depend on mutable state (e.g., reading a class field). To maintain verification soundness, Dafny requires functions to declare their *read frame* using the `reads` clause (Code Segment 2.12).

Code Segment 2.12. Reads Clause

```
class Sensor {
   var value: int

   function GetValue(): int
      reads this // Result depends on the state of 'this' object
   {
      this.value
   }
}
```

Framing: Modifies Clauses

The `modifies` clause is the imperative counterpart to `reads` (Code Segment 2.13). It strictly limits the scope of memory mutations a method can perform. Any object not

listed in the `modifies` clause is guaranteed to remain unchanged. This strict framing is essential for *modular verification*, allowing callers to reason about method side effects without analyzing the method's body.

Complex updates, such as iterating over an array of objects, require granular specification. Dafny permits set comprehension syntax within `modifies` clauses to dynamically specify sets of modified objects.

Code Segment 2.13. Modifies in Loops

```
class Container {
  var items: array<Item>

  method UpdateAll()
    // Frame: Validates that this method alters the objects inside the
        ↪ array
    modifies this.items[..]
  {
    for i := 0 to items.Length
      // Loop Frame: The loop body only modifies the i-th item
      modifies this.items[i]
    {
      this.items[i].Update();
    }
  }
}
```

Termination and Decreases

Dafny verifies that recursion and loops terminate. To prove termination for loops and recursion, Dafny uses a *termination metric*, specified manually by the `decreases` clause (Code Segment 2.14). This metric must represent a well-founded ordering that strictly decreases with each step. Dafny often infers this metric automatically, but complex recursions and loops may require an explicit `decreases` clause.

Code Segment 2.14. Decreases Clause

```
function Ackermann(m: int, n: int): int
   requires m ⩾ 0 && n ⩾ 0
   // Lexicographic ordering: strictly decreases on m, or on n if m
      ↪ stays same
   decreases m, n
{
   if m == 0 then n + 1
   else if n == 0 then Ackermann(m - 1, 1)
   else Ackermann(m - 1, Ackermann(m, n - 1))
}
```

Two-State Expressions

Functional correctness often involves relating the state of a object *after* execution to the state *before* execution. Dafny provides specific operators for these two-state specifications (Code Segment 2.15):

- `old(e)`: Value of expression `e` at the start of the calling method.
- `unchanged(o)`: A predicate asserting that the fields of object `o` have not changed. It is semantically equivalent to checking `old(o) == o` in the postcondition of a method.
- `fresh(o)`: Asserts that object `o` was (re)allocated during the current method. This is crucial for proving that objects are unique (non-aliased).

Code Segment 2.15. Two-State Expressions

```
class Counter {
   var count: int

   constructor()
      ensures fresh(this) // 'this' is a new allocation
      ensures count == 0
   {
      count := 0;
   }

   method Increment()
      modifies this
      ensures count == old(count) + 1 // Relates post-state to pre-state
   {
      count := count + 1;
   }
}
```

2.7.3 Dafny Standard Libraries

To aid in the development of verified software, Dafny offers a standard library of formally verified data structures and functions (Code Segment 2.16). These libraries reduce verification time and effort by providing common datatypes and proofs.

For example, the `Std.Wrappers` module provides an optional (`Option<T>`) and error handling (`Result<T>`) datatype, similar to other standard programming languages. Similarly, `Std.Collections.Seq` provides common verified operations on sequences.

Code Segment 2.16. Standard Libraries

```
import Std.Collections.Seq
import Std.Wrappers

method UseLibraries() {
   // Verified sequence manipulation
   var s1 := [1, 2, 3];
   var s2 := Seq.DropFirst(s1); // s2 == [2, 3]

   // Verified Option types
   var valid: Wrappers.Option<nat> := Wrappers.Option.Some(10);
   assert valid.Some? && valid.value == 10;
}
```

CHAPTER 3

Literature Review

NoC research and development has been active for the past several decades, largely due to the need for efficient communication methods across multi-core chips consisting of several processing cores, caches, accelerators, and I/O [9]. Research in PMC and SMC has produced numerous tools, algorithms, and case studies. Finally, research into assume-guarantee and modular reasoning has seen increased focus, promising greater scalability and compositionality for formal verification. This section explores the existing literature in these areas, and describes the areas from which this thesis is derived and identifies the research gaps it aims to address.

3.1 NoC Verification

Formal techniques have been used to verify the correctness of many NoC designs at varying levels of abstraction and with many different tools; see [37] for an overview of recent work. Informal or semi-formal methods are lightly covered inasmuch as they motivate this thesis.

3.1.1 Functional NoC Verification

Model checking provides largely automated guarantees for NoC design, but is limited by the state-space explosion problem and is not scalable to larger or more complex models. Zaman et al. [38] verify deadlock freedom of a programmable NoC design with eight nodes and a single router using the SPIN model checker [15]. Their approach acknowledges the state-space limitations of model checking, and does not cover common and more complex NoC designs where there are many routers. Dridi et al. prove that their NoC router, DAS, satisfies the safety and performance properties required for *Mixed-Criticality Systems* (MCSs) using the IF language [39]. While DAS was verified correct, the number of

processes verified had to be limited in order to prevent state-space explosion. The CADP toolbox [40] allows for verification of deadlock freedom and other temporal properties, as well as facilities for generating test vectors. Zhang et al. used CADP to prove livelock freedom of a fault-tolerant routing algorithm on an asynchronous NoC, but were not able to scale this verification larger than a 2×2 mesh NoC [41]. The lack of scalability of model checking is impractical for NoC verification, as real-world mesh NoC designs are commonly implemented at sizes from 2×2 up to 64×64 [9]. This motivates the use of deductive verification for functional correctness of NoC design.

Interactive Theorem Provers (ITPs) offer a scalable way to prove functional correctness of NoC designs, but typically require explicit user guidance, specialized knowledge, and significant manual proof effort. Despite this, there exist many case studies of ITPs verifying NoC designs. GeNoC, a generic NoC metamodel created in the ACL2 [34] ITP is a framework for constructing proofs of functional correctness of NoC designs [3] and has proven deadlock freedom of several routing algorithms, as well as functional correctness of several diverse NoC implementations [42–45]. Taylor and Zhang use *property directed reachability* (PDR) and the IVy [46] verification tool to scale the livelock freedom from [41] to arbitrarily sized NoCs. Strictly speaking, PDR is not ITP, but rather a fully automated verification method implemented in IVy. These methods are powerful, but are inaccessible without domain expertise in ITPs.

Both model checking and theorem proving are effective in NoC verification. However, these approaches often require significant effort and knowledge in specialized modeling languages (e.g., Promela in SPIN) or industrial ITPs (e.g., ACL2). This required knowledge presents a hurdle to formal NoC verification, suggesting a research gap for a verification approach that uses a programming language that can represent both programs and hardware models. This thesis fills this gap by specifying a NoC model using the Dafny language [36] (see Section 3.3).

3.1.2 Quantitative NoC Verification

Analytical methods based on queueing theory [47], *Probabilistic Timing Analysis* (PTA)

[48], and evolutionary algorithms [49] can estimate metrics such as latency and power usage, but lack the strict guarantees provided by formal analysis. Kiasari et al. estimate expected packet latency and throughput using a queueing theory abstraction [47]. Their tool generates results four orders of magnitude faster than traditional simulation, but diverge from the actual results in certain circumstances and do not provide formal guarantees of performance. Alhubail and Bagherzadeh present an approach inspired by formal methods utilizing the *Strength Pareto Evolutionary Algorithm2* (SPEA2) to maximize performance and minimize power usage in a NoC connected to a heterogeneous CPU-GPU processor [49]. Their work is effective in optimizing processor placement and buffer size in a NoC, but still relies on simulation-driven results lacking formal guarantees. Slijepcevic et al. propose a tree-based NoC that uses PTA to derive a tight *Worst-Case Execution Time* (WCET) estimate, but this time is only an estimate, without any formal bounds or statistical certainty relating to a formal model [48]. This lack of formal guarantees motivates the use of formal methods capable of quantitative analysis, such as PMC, early in the NoC design cycle.

Formal quantitative methods have been applied to NoC design, but existing approaches are either limited in scope or contain critical gaps in scalability or semantic modeling. Sharifi et al. optimize routing algorithm selection in mesh-style NoCs using model checking and *Timed Automata* (TA) to quantify the predicted network latency of each routing algorithm [50]. Their work demonstrates the potential for highly accurate quantitative information from formal methods, but only scales to 4×4 NoCs. Tatas uses DTMCs to model the expected number of hops from source to destination in bufferless NoCs in the presence of deflection routing [51]. This work shows that Markovian models can be used to model NoCs, but only verifies bufferless NoCs which are not used in any of the eight real-world NoCs observed in [9]. Lewis et al. characterize PSN on an abstract mesh NoC model using the MODEST TOOLSET [23] and show how traffic patterns influence PSN [6]. Roberts et al. extend the PSN analysis work of Lewis et al. [6] to a concrete 2×2 mesh NoC and derive a traffic injection pattern that reduces PSN across the NoC [8]. Their work is restructured and extended by Boe with a framework for performing probabilistic

verification of a *modular* NoC model [7]. The framework provides a parameterized, modular NoC model within the MODEST TOOLSET; however, the synchronous digital semantics of a NoC are not correctly modeled, leading to an unnecessary state-space explosion with many redundant interleaving states. This flaw rendered formal PMC analysis intractable, forcing the use of SMC to characterize PSN [7]. This reveals two distinct shortcomings: (1) the underlying model is semantically flawed and unoptimized, and (2) the SMC-based analysis was only demonstrated for 2×2 and 3×3 mesh NoCs, leaving its application to larger, more realistic networks unexplored. These shortcomings motivate both the optimization of the NoC framework from [7] to correct the semantic model and a thorough evaluation of the framework's scalability for more diverse sizes and parameters.

Alternative formal methods for performing probabilistic verification of digital systems involve using *Stochastic boolean SATisfiability* (SSAT) solvers [52] and *Propositional Model Counting* (#SAT) [53]. Lee et al. [54] combine SSAT, #SAT, and PMC to provide an efficient solver toward probabilistic design of digital systems. These methods are promising, but focus on gate-level logical verification, while this thesis addresses quantitative properties at an architectural level of abstraction.

3.2 Assume-Guarantee Reasoning

AG reasoning is an established, compositional technique for overcoming state-space explosion in the formal verification of large, complex systems. For example, Pnueli et al. describe a method for verifying classes of parameterized systems consisting of n processes using AG reasoning and finite state model checking demonstrating that the state-space explosion problem can be reduced even with existing finite-state model checking tools [55]. Abdulla et al. build on the work of Pnueli et al. [55] with a more efficient method of automatically verifying parameterized systems using compositional reasoning [56]. McMillan used AG reasoning to render the verification a bit-level implementation of an "out-of-order" microprocessor based on Tomasulo's algorithm [57] tractable [58]. Henzinger et al. overcome the computational limitations of verifying a parallel *digital signal processing* (DSP) chip with 64 compute cores by applying AG principles to each individual core and then

composing those verified cores together [59]. Their approach uncovered several previously unknown bugs in design, and kept the verification tractable [59]. McMillan and Zuck develop a compositional testing and verification framework for the QUIC [60] internet protocol, demonstrating how to convert global correctness formulas into local formulas suitable for AG reasoning [61].

Crucially, this compositional framework has been successfully extended to probabilistic verification, demonstrating its ability to make intractable PMC problems tractable. Kwiatkowska et al. describe a formal method for probabilistic AG reasoning, and demonstrate that their method can verify systems where traditional PMC was intractable [62]. Nuzzo et al. develop a framework for probabilistic AG reasoning for closed-loop control with probabilistic requirements for *Cyber-Physical Systems* (CPSs) and demonstrate the effectiveness of their approach on an aircraft power distribution network [63]. Calinescu et al. demonstrate the usage of probabilistic AG reasoning in the reverification process of large-scale distributed IT projects such as cloud infrastructure [64]. While probabilistic AG is a proven technique for scaling quantitative analysis, it has not yet been applied to the domain of NoC PSN characterization. This thesis will explore formal decomposition of the NoC model from [7], which in the future could lead to modular PSN characterization of arbitrary $n \times n$ NoCs.

3.3 System Verification Using Dafny

The Dafny programming language [36] implements the modular Hoare logic [11] and provides an interactive verification environment. The core of Dafny is built on AG reasoning: the user provides assumptions and guarantees of a method (or function) and then must prove that the implementation of that method matches the AG specification. After doing so, the method can be called in other areas. When it is called, only the AG specification is checked, and the implementation details are abstracted away. This provides a scalable way to verify complex program functionality.

Leino [65] demonstrates Dafny's utility for verifying concurrent systems. By developing safety and liveness properties for a concurrent ticket system—drawing analogies to proof

systems like TLA$^+$ [17] and Event-B [66]—Leino shows that Dafny is capable of proving complex temporal properties, not just sequential program correctness. This thesis will use this demonstration of Dafny's reasoning power to verify NoC designs.

METHODS

The following chapters outline the main research and engineering presented in this thesis. Chapter 4 describes the updates to the modular NoC model from [7]. Chapter 5 covers the CTL verification of the NoC models using the MODEST TOOLSET and the generation of additional results of the NoC models using the MODEST TOOLSET. Chapter 6 cover the verification of the NoC model using the verification-aware Dafny programming language.

CHAPTER 4

Expansion of the Modular NoC Model

This chapter presents a refined formal NoC model and discusses the approach for abstracting the model for scalable verification.

4.1 Modular Design of the Formal NoC Model

Previous work [7] provided a general framework for creating arbitrary $n \times n$ mesh-style NoCs in the MODEST language. The NoC model presented by [7] consists of a parallel composition of unique router process instances. This *modular* style was influenced by modern digital design using *hardware description languages* such as Verilog [67] or VHDL [68], where modular design is essential for scalable designs and commonly used across all hardware designs. A *modular design* is defined by the clean separation of circuitry between different modules, where each module has a well defined interface that can be connected to other modules. We term the model from [7] as modular, as all logic is shared between router instances and each router has a clear interface to other routers. This section details the updates to the modular NoC model completed for this thesis.

4.2 Data Structures and Types

We define a set of custom datatypes to formally specify the state of the NoC. These types are designed to balance expressiveness with state-space manageability.

4.2.1 Packet Datatype

Consistent with Definition 2.1, we abstract a packet as a scalar value representing the identifier of its destination router. For an $n \times n$ NoC, the packet type is defined as a bounded integer constrained to the domain $[0, n^2 - 1]$, as shown in Code Segment 4.1. The bounded definition assists in maintaining a small state space during model checking.

Code Segment 4.1. Packet Datatype

```
const int NOC_MAX_ID = 9; /* for a 3x3 */
int(0..NOC_MAX_ID) packet;
```

4.2.2 Buffer Datatype

To support arbitrary buffer depths, we model buffers as recursive *Algebraic Data Types* (ADTs). This structure, shown in Code Segment 4.2, functions as a standard FIFO queue. The recursive definition allows the buffer capacity to be parameterized globally and verified against overflow properties without altering the underlying data structure definition or changing its operational semantics.

Code Segment 4.2. `buffer` Datatype

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

The operational semantics for the buffer (Definition 2.2) are defined in Code Segment 4.3. `enqueue` and `dequeue` manage buffer expansion and reduction, respectively. To facilitate the specification of temporal properties later in this work (e.g., verifying that a specific packet is in a buffer), we include a recursive `contains` predicate. `peekFront` returns the element at the front of the buffer, or -1 if the buffer is empty which helps catch invalid packets.

Code Segment 4.3. `buffer` Operations

```
function buffer option enqueue(int n, buffer option ls) =
   some(buffer {
      hd: n,
      tl: ls
   });

function buffer option dequeue(buffer option ls) =
   if ls == none then none
   else if ls!.tl == none then none
   else some(buffer {
      hd: ls!.hd,
      tl: dequeue(ls!.tl)
   });

function bool contains(int id, buffer option ls) =
   if ls == none then false
   else if ls!.hd == id then true
   else contains(id, ls!.tl);

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

4.2.3 Flagged Buffer Datatype

Capturing the cycle-accurate behavior of a digital FIFO buffer using only the recursive ADT described in Section 4.2.2 is challenging within the async MODEST language. To ease the enforcement of synchronous digital semantics, we encapsulate the list in a wrapper type, `flaggedBuffer` (Code Segment 4.4).

This structure explicitly decouples the buffer's content from its cycle-accurate observable status flags (`isEmpty`, `isFull`, `serviced`). This separation prevents race conditions (e.g., write-before-read conflicts) by ensuring that status flags reflect the state at the beginning of a clock cycle, regardless of interleaved updates that may occur during each the clock cycle.

Code Segment 4.4. `flaggedBuffer` Datatype

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

4.3 The Router Model

The `Router` process is the top-level unit used for NoC composition as described in Definition 2.4. It encapsulates the control logic (arbitration and routing) and the structural state updates required to model the generic router architecture depicted in Figure 2.2.

4.3.1 Router Datatype

The `router` datatype (Code Segment 4.5) defines the state for a single router node (Definition 2.3). To minimize state-space explosion, we exclude the static topology data described in [7] (e.g., neighbor IDs) from the router state, relying instead on the stateless topology calculation described in Section 4.4. The router state consists of:

- **I/O State:** Five `flaggedBuffer` instances representing the four cardinal directions and the local direction ($\mathcal{B}$ and $\mathcal{C}$ in Def. 2.3).
- **Control State:** Variables maintaining the round-robin arbitration history and current servicing pointers ($\mathcal{A}$ and $\mathcal{L}$ in Def. 2.3).
- **PSN Intrumentation:** Integer counters (e.g., `thisActivity`) for tracking PSN events.

The member variables for the router datatype are shown in Table 4.1.

Member Variable	Description
`buffers`	Array of `flaggedBuffer` instances managing packet input/output.
`priority_list`	Ordered list of input directions determining the service priority for the current clock cycle.
`priority_list_temp`	Temporary array used for computing the next priority permutation. This array is present in the router state due to the implementation of priority arbitration. The priority arbitration is done across several processes, and data sharing between the processes is done through global state variables.
`serviced_index`	Pointer to the next buffer to be serviced in the current cycle.
`unserviced_index`	Pointer tracking buffers that sent no packets this cycle.
`total_unserviced`	Counter of buffers that failed to send, used for computing the next priority permutation.
`thisActivity`	Cumulative activity counter for the current clock cycle.
`lastActivity`	Activity count from the previous cycle (used for inductive PSN analysis).
`used`	Boolean flags indicating which channels were active this cycle.

Table 4.1: Router State Member Variables

Code Segment 4.5. `router` Datatype

```
datatype router = {
   flaggedBuffer[] buffers,
   int(0..4)[] priority_list,
   int(0..4)[] priority_list_temp,
   int(0..4) serviced_index,
   int(0..4) unserviced_index,
   int(0..5) total_unserviced,
   int thisActivity,
   int lastActivity,
   bool[] used
};
```

4.3.2 Buffer State Sampling (PrepRouter)

To model synchronous digital semantics within the asynchronous MODEST semantics, we use a `PrepRouter` process as presented in [7] (Code Segment 4.6). This process effectively "samples" the state of input buffers at the rising edge of the virtual clock.

By setting the `isEmpty` and `isFull` flags before any routing or arbitration logic executes, we prevent intra-cycle causality violations (where a flit traverses multiple routers in a single cycle). This ensures the MODEST model adheres to the digital semantics where outputs depend only on the state of circuit at the beginning of a clock cycle.

Code Segment 4.6. `PrepRouter` Process

```
action prepRouter;
process PrepRouter(int id) {
 // Set channel state at the clock edge
 prepRouter {=
  // North Channel (0)
  noc[id].channels[NORTH].isEmpty
    = len(noc[id].channels[NORTH].buffer) == 0,
  noc[id].channels[NORTH].isFull
    = isBufferFull(noc[id].channels[NORTH].buffer),

  /* ... repeated for channels 1-4 ... */
 =}
}
```

4.4 NoC Topology and State

The entire state of the NoC is represented by a single array of router instances, indexed by router identifier. This array represents $\mathcal{R}$ from Definition 2.4, while $\mathcal{C}$ is covered in Section 4.4. Global visibility is required because MODEST does not support passing references to sub-process calls, and each router must be able to access its neighbors states to send packets. The NoC data structure is defined in Code Segment 4.7.

Code Segment 4.7. NoC Data Structure

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

Initialization Strategy

The NoC state must be initialized before router operations can occur. Additionally, validating CTL properties requires that the model satisfies the property in all reachable states, including the initial state. Consequently, not only does the `noc` array need to be initialized

before router operations begin, but it also must be initialized in the initial state of the system. The MODEST language allows for dynamic variables, like arrays, to be unitialized, and the model from [7] used this to simplify initialization logic. This delayed initialization approach works with the PSN characterization properties presented in Section 5.1, but does not work with the newly developed CTL properties in Section 5.2. Because of this, we developed a new initialization method for the model.

Attempt 1: Static Initialization. Our initial initialization approach utilized an external Python script to generate the static initial state for a specific $n \times n$ topology. This script existed because the model initially lacked the logic to derive the identifiers of neighboring routers dynamically in MODEST. The script calculated the `ids` array from [7] for each router and hard-coded it into the initial NoC state (Code Segment 4.8). These neighbor identifiers correspond to $\mathcal{C}$ in Def. 2.4 and indicate a connection between two routers.

Code Segment 4.8. Static Initialization (Deprecated)

```
router[] noc = [
router {
   /* ... channels ... */
   ids: [NO_CONNECT, NO_CONNECT, 1, 2], // Static neighbor references
   /* ... remaining state ... */
},
/* ... continued for each router ... */ ]
```

While functional, this approach introduced significant limitations:

1. **External Dependency:** The model was not self-contained, e.g., changing the mesh size required regenerating parts of the model code via the external script.
2. **State Redundancy:** The `ids` array used memory in every router instance to store static topological data that could be computed dynamically. At the time of writing, MODEST does not allow for constant arrays so each `ids` array was stored in every state vector, greatly increasing the state-space.

These limitations damage the modularity and verification memory usage of the NoC model, and are a barrier to scalable verification with this model.

Attempt 2: Dynamic Neighbor Identifier Calculation. To address these limitations and create a truly modular NoC, we replaced the static `ids` array with a dynamic neighbor identifier calculation function. Derived from the stateless neighbor logic developed in Section 6.27, this function leverages the regular mesh-style geometry of the NoC to compute neighbor identifiers on the fly.

This optimization demonstrates the classic time-memory trade-off in model checking. Looking up a neighbor now requires several arithmetic operations (increasing computational overhead), but it eliminates the `ids` state vector entirely. Given that *state-space explosion* is the primary bottleneck during our model checking, reducing the memory required by each state is the best design choice.

The implemented logic (Code Segment 4.9) validates router bounds and computes neighbor identifiers based on the mesh-style layout of the NoC.

Code Segment 4.9. Neighbor Identifier Calculation

```
function bool isValidID(int id) = 0 ≤ id && id ≤ NOC_MAX_ID;

function int getNeighborID_internal(int id, int dir) =
   if dir == NORTH then id - NOC_MESH_WIDTH
   else if dir == SOUTH then id + NOC_MESH_WIDTH
   else if dir == EAST && (idToColumn(id) < NOC_MESH_WIDTH - 1) then id
      ↪ + 1
   else if dir == WEST && (idToColumn(id) > 0) then id - 1
   else NO_CONNECT;

function int getNeighborID(int id, int dir) =
   if isValidID(getNeighborID_internal(id, dir))
   then getNeighborID_internal(id, dir)
   else NO_CONNECT;
```

With this logic in place, the NoC state can be initialized procedurally using MODESTs `array` constructor, regardless of NoC size (Code Segment 4.10).

Code Segment 4.10. Dynamic Default Initialization

```
router[] noc = array(i, NOC_MAX_ID + 1, // NOC_MAX_ID + 1 = n^2
router {
   buffers: [
      flaggedBuffer {buffer: none, /* flags initialized */},
      /* ... repeated for each buffer ... */],
   /* 'ids' removed: Neighbor data is now computed statelesssly */
   priority_list: [NORTH, EAST, SOUTH, WEST, LOCAL],
   priority_list_temp: [0, 0, 0, 0, 0],
   serviced_index: 0,
   unserviced_index: 0,
   total_unserviced: 0,
   thisActivity: 0,
   lastActivity: 0,
   used: [false, false, false, false, false]
});
```

4.4.1 NoC Construction using Parallel Composition

The complete $n \times n$ NoC is defined as the parallel composition of n^2 unique `Router` processes and a global `Clock` process. Code Segment 4.11 demonstrates the composition of a 2×2 mesh. Note that unlike previous modular model [7], this composition relies on actions inside each sub-process called by `Router` (Section 4.5.3) to maintain synchronization as opposed to actions directly in the `Router` process.

Code Segment 4.11. Parallel Composition of a 2×2 NoC

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

4.5 Execution Model and Synchronization

This section details the formal execution model of the NoC, focusing on the additional synchronization required to better align the asynchronous semantics of Modest with the synchronous nature of digital logic (Sec. 2.1) and reduce the state-space of the model. We demonstrate how extending the synchronization method from [7] enforces cycle-accurate behavior and significantly reduces the state space needed for CTL verification.

4.5.1 Synchronization Challenges in Parallel Composition

While the modular architecture introduced in [7] improved scalability via the parallel composition of router processes (Code Segment 4.11), it introduced parallel interleavings of router execution inherent to asynchronous process algebras. Unlike the monolithic model in [8], where a single process orchestrated state updates, the parallel composition $\mathcal{N} = R_0 \parallel \cdots \parallel R_{n^2-1} \parallel \mathit{Clock}$ permits asynchronous interleavings that violate the digital semantics. We term this *temporal drift*, where one router process may execute multiple cycles while others execute none. This in turn causes a *causality violation*, where a packet could traverse multiple routers in a single clock cycle if the `AdvanceRouter` process of R_i precedes the `PrepRouter` action of R_{i+1}. This behavior is physically impossible in a synchronous, clocked circuit.

Previous work [7] removed this erroneous behavior with a two synchronization actions (barriers), `tick` and `tock`, that constrained the asynchronous interleavings of each router. However, these actions were insufficient to limit the state space enough to accomplish model checking of the modular model, even for a small 2×2 NoC. The asynchronous interleaving of intermediate assignments (e.g., updating buffers, updating pointers) caused a combinatorial explosion in the state space. These asynchronous interleavings are not present in a synchronous digital model, and thus should not have any effect on the operation of the NoC. For CTL model checking to be tractable for 2×2 models, we need to enforce lock-step execution of assignments in each router not just in two places, but for every intra-cycle assignment.

4.5.2 Barrier Synchronization via Shared Actions

To eliminate these spurious interleavings and more faithfully model synchronous digital semantics, we exploit the multi-way synchronization capability of MODEST *actions*. By defining a unique action for each assignment in the router's execution loop, we create a global synchronization between each state update in the NoC.

Consider the parallel assignment example in Code Segment 4.12. In a standard asynchronous composition, the model checker explores all permutations of updates (s_1, s_2 in

Figure 4.1). By decorating the update with an action (`clock`) in Code Segment 4.13, we force the assignment to occur simultaneously for both processes. This collapses the intermediate interleaving states into a single transition that mimics the synchronous update of a digital system.

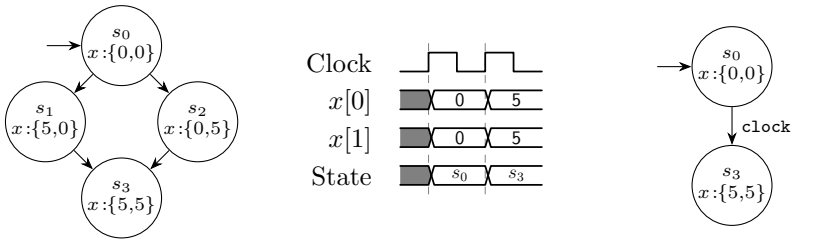
Code Segment 4.12. Parallel Composition Interleaving

```
int x[] = {0, 0};
/* Asynchronous: Explores x[0]=5 → x[1]=5 AND x[1]=5 → x[0]=5 */
process setTo5(int id) {{= x[id] = 5; =}}
par { :: setTo5(0) :: setTo5(1) }
```

Code Segment 4.13. Synchronous Composition

```
action clock;
int x[] = {0, 0};
/* Synchronous: Explores only {x[0]=5, x[1]=5} simultaneously */
process setTo5(int id) { clock{= x[id] = 5; =} }
par { :: setTo5(0) :: setTo5(1) }
```

We exploit these synchronizing actions at a larger scale in the modular NoC model to reduce the state-space of the NoC model and simultaneously more closely model the expected digital semantics.



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

Fig. 4.1: State Space of a Simple Parallel Composition

4.5.3 The Router Execution Loop

The `Router` process is structured as a phased execution loop, where each assignment

in each phase is guarded by a synchronizing action. This ensures that every router in the NoC mesh executes the same assignment phase simultaneously.

Code Segment 4.15 defines the top-level behavior. The router cycles through five distinct phases:

1. **Generation (`gf`):** New packets are injected into local buffers.
2. **Preparation (`pr`):** Input states are saved for the current clock cycle (see Section 4.3.2).
3. **Advancement (`ar`):** Packets are routed to output channels.
4. **Arbitration (`up`):** Priority pointers are updated.
5. **Instrumentation (`gn`):** Noise metrics are aggregated.

Code Segment 4.14. Synchronization Barriers

```
action gf, pr, ar, up, gn;
/* Sub-processes use these actions to enforce lock-step */
```

Code Segment 4.15. `Router` Process

```
process Router(int id) {
   GenerateFlits(id);   /* Barrier: gf */
   PrepRouter(id);      /* Barrier: pr */
   AdvanceRouter(id);   /* Barrier: ar */
   UpdatePriority(id);  /* Barrier: up */
   updateThisActivity(id); /* Barrier: gn */

   nextClockCycle;      /* Global Clock Barrier */
   updateLastActivity(id); /* Barrier: gn */
   Router(id)           /* Recursive Call */
}
```

4.5.4 The Clock Process

The `Clock` process (Code Segment 4.16) serves as another synchronizer. It participates in the `nextClockCycle` action, ensuring that the clock cycle advances only after all routers have completed all of their intra-cycle phases.

At a first glance, this process looks to be erroneous, as it models recursion without a

end condition. However, this is the behavior that we want. The NoC model has a finite state space, but should be able to represent infinite execution paths (although is practice they are always bounded by floating point precision). This infinite recursion works in MODEST because there is no state associated with each transition. `clk` is a transient variable, so it only updates during a transition, and is not stored in the state space. In this way we model infinite execution paths in the finite space of the NoC model.

The variable `clk` is tracked as a transient variable to support bounded temporal properties. This transient variable becomes a transition reward in the underlying DTMC model (Section 2.5.3). For example, verifying "a packet is generated within K cycles" requires referencing the accumulated sum of the `clk` transition reward.

Code Segment 4.16. `Clock` Process

```
transient int clk;
process Clock() {
   // Advances time only when all Routers are ready
   nextClockCycle {= clk = 1 =};
   Clock()
}
```

4.6 Traffic Injection (GenerateFlits)

The modular `GenerateFlits` process functions as the *stimulus generator* for the formal model. It contains the stochastic logic that determines when and where new packets are introduced into the network and the destination of each packet. By decoupling this logic from the router architecture, we can evaluate the PSN impact of various traffic patterns without altering the underlying execution model.

4.6.1 Packet Injection Process

The packet injection process (Code Segment 4.17) executes as the first phase of the router's synchronous loop and models the behavior of a PE adding new packets to the network. The process accepts a router identifier `id` and conditionally enqueues a new

packet into the local buffer $R_{id}.B_L$, subject to backpressure in the router (i.e., the local buffer must not be full).

Crucially, this process adheres to the barrier synchronization described in Section 4.5.2. Whether a new packet is generated or not, the process *must* execute the `gf` action. This action ensures that the state of each router transitions in lock-step, preserving the cycle-accurate semantics of the model and limiting interleavings.

Code Segment 4.17. `GenerateFlits` Process Template

```
action generateFlits;
process GenerateFlits(int id) {
   if (/* injection condition met */ && /* local buffer has space */) {
      /* State Update: Enqueue new packet */
      generateFlits {= ... =}
   } else {
      /* No-Op: Still synchronize with global barrier */
      generateFlits
   }
}
```

4.6.2 Uniform Traffic Pattern

Code Segment 4.18 implements a uniform injection rate used for baseline verification [7, 8]. This pattern injects flits at a fixed rate of three new packets per ten clock cycles.

The destination logic uses the *self-excluding uniform distribution* described in [7]. A destination router identifier is selected from the range $[0, n^2 - 2]$. If the selected value equals or exceeds the source router `id`, it is incremented. This maps the sample space to the set of all routers excluding the source, ensuring destination is not equal to id. Additionally, this provides a uniform probability of generating a packet destined for every router except the source. This is shown as the assignment prefaced with `1:` in Code Segment 4.18.

Code Segment 4.18. Uniform Random Injection (30% Load)

```
process GenerateFlits(int id) {
   int(0..NOC_MAX_ID) destination;
   /* Injection Condition: Buffer space available AND periodic cycle
      ↪ check */
   if (!isBufferFull(noc[id].channels[LOCAL].buffer) && clk % 10 < 3) {
      generateFlits {=
         0: destination = DiscreteUniform(0, NOC_MAX_ID - 1),
         /* Self-Exclusion Logic: Map [0, N-1] to [0, N] {id} */
         1: noc[id].channels[LOCAL].buffer =
            enqueue(destination ⩾ id ? destination + 1 : destination,
                  noc[id].channels[LOCAL].buffer)
      =}}
   else { generateFlits }
 }
```

4.6.3 Bursty Traffic Pattern

To evaluate NoC performance under a different packet injection pattern, we model "bursty" traffic (Code Segment 4.19). This model utilizes two state variables, `burst_times` and `sleep_times`, to toggle each router between an *active state* (continuous injection) and a *dormant state* (no injection).

The duration of each phase is determined at runtime. When a router transitions state, it samples a duration from a uniform distribution defined by constant bounds (e.g., `BURST_MAX`). This allows the model to explore cases where multiple routers saturate the network simultaneously, followed by periods of recovery and cases where each router injects packets at a different time while the others are sleeping.

Code Segment 4.19. Stochastic Bursty Injection

```
/* Global state tracks phase duration for each router */
int[] burst_times = array(i, NOC_MAX_ID + 1, 0);
int[] sleep_times = array(i, NOC_MAX_ID + 1, 0);
/* Configuration constants for phase lengths */
const int BURST_MIN = 10, BURST_MAX = 100;
const int SLEEP_MIN = 200, SLEEP_MAX = 400;

process GenerateFlits(int id) {
```

```
    int(0..NOC_MAX_ID) dest;
    if (isBufferFull(noc[id].channels[LOCAL].buffer)) {
       generateFlits /* Backpressure: Skip injection but sync */
    } else {
       /* Active Phase: Inject and decrement burst counter */
       if (burst_times[id] != 0) { generateFlits {=
          0: dest = DiscreteUniform(0, NOC_MAX_ID - 1),
          1: noc[id].channels[LOCAL].buffer =
            enqueue(dest ≥ id ? dest + 1 : dest, noc[id].channels[LOCAL
               ↪ ].buffer),
          2: burst_times[id] = burst_times[id] - 1
       =}}
       /* Dormant Phase: Decrement sleep counter */
       else if (sleep_times[id] != 0) { generateFlits {=
          sleep_times[id] = sleep_times[id] - 1
       =}}
       /* Phase Transition: Sample new durations */
       else { generateFlits {=
          burst_times[id] = DiscreteUniform(BURST_MIN, BURST_MAX),
          sleep_times[id] = DiscreteUniform(SLEEP_MIN, SLEEP_MAX)
       =}}
}}
```

4.7 Routing, Arbitration, and PSN Instrumentation

This section details the implementation of the routing logic, arbiter, and the instrumentation added to track PSN. While the functional logic mirrors the baseline in [7], we introduce synchronization barriers (Section 4.5) and more granular activity tracking to improve PSN analysis.

4.7.1 Routing Preparation (PrepRouter)

The first step of the routing logic in the `Router` process (Code Segment 4.15) is `PrepRouter`. This process sets the status flags about a buffer (`isEmpty`, `isFull`) as a part of modeling the synchronous NoC behavior in the asynchronous MODEST language. Once these flags are set they remain set for the entirety of the clock cycle, which correctly models the synchronous digital behavior.

4.7.2 X-Y Routing (AdvanceRouter)

The routing algorithm determines the path of a packet from source to destination. Our modular design encapsulates this logic in the `AdvanceRouter` process, allowing the routing algorithm to be swapped without altering the other functional components of the router model. As described in [7], we implement *Dimension-Order Routing* (specifically X-Y routing) as the default algorithm due to its deadlock-free properties and minimal logic overhead [9, Ch. 4]. Code Segment 4.15 shows where `AdvanceRouter` is in the top level `Router` process.

Hierarchical Execution

Routing executes hierarchically. The `AdvanceRouter` process (Code Segment 4.20) iterates through the input buffers in order according to the current `priority_list`, as done in [7]. For each buffer, it invokes `AdvanceChannel`, which attempts to forward the head packet toward its destination.

Code Segment 4.20. `AdvanceRouter` Process

```
process AdvanceRouter(int id) {
   /* Iterate through inputs in priority order */
   AdvanceChannel(id, noc[id].priority_list[0]);
   AdvanceChannel(id, noc[id].priority_list[1]);
   AdvanceChannel(id, noc[id].priority_list[2]);
   AdvanceChannel(id, noc[id].priority_list[3]);
   AdvanceChannel(id, noc[id].priority_list[4])
}
```

Formal Logic

Algorithm 1 presents the formal definition of the X-Y routing logic. The algorithm starts by checking if the source buffer b has packets (line 2). If it does not, the algorithm is finished as there is nothing to route.

In the case that b is not empty, we look at the front packet available in b (line 4). If this packet is destined for the current router, then we dequeue the packet and return, modeling the PE consuming it.

In the case that b has a packet that is not destined for the current router, we must execute X-Y routing. Routing starts by calculating the direction the packet must travel. This is done by first calculating the change in X (Δx) and Y (Δy) that the packet must make to arrive at it's destination. Lines 12-15 show how these deltas are transformed into directions, and additionally show how X-Y routing executes movements in the X direction first with Δx being checked before Δy.

One line 16 we do not have an *else* condition because in that case Δx and Δy would be equal to 0. If they are both equal to 0 then the packet has arrived at the destination, and therefore the code path starting at line 6 would have executed.

After we have calculated the direction, we get the identifier of the neighboring router (line 17). This can be statically calculated because the dimension of the NoC is fixed at runtime.

Lines 18-23 show the code path of moving the packet from the current router to the neighboring router. This process can only take place if the receiving buffer of the neighboring router is not full. If it is not full, the router dequeues the packet from its buffer, and enqueues it onto the neighbors buffer. The activity counters are then updated, and the buffer is marked as being serviced in the current cycle.

4.7.3 Round-Robin Arbitration (UpdatePriority)

To prevent starvation, the router employs a Round-Robin arbiter. The `priority_list` determines the order in which input buffers are checked (Code Segment 4.20).

The arbitration logic tracks which buffers were *serviced* (successfully transmitted a packet) in the current cycle. In the `UpdatePriority` phase, the priority list is rotated such that the most recently serviced buffer is moved to the lowest priority, ensuring fair access to output channels over time. This logic is unchanged from [7] but is now guarded by a synchronization action (see Code Segment 4.15 to see where this logic executes in the router).

4.7.4 Per-Router PSN Instrumentation

Algorithm 1 Formal X-Y Routing Logic

Require: Router id, Source Buffer b
Ensure: Packet sent to neighbor or PE $\vee$ buffer state unchanged

1: **procedure** AdvanceChannel(id, b)
2: **if** $isEmpty(b)$ **then return**
3: **end if**

4: $p \leftarrow peek(b)$
5: **if** $p.dest == id$ **then** ▷ Reached destination, send to PE
6: $b \leftarrow dequeue(b)$
7: $noc[id].thisActivityTemp \leftarrow noc[id].thisActivityTemp + 1$
8: **return**
9: **end if**

10: $\Delta x, \Delta y \leftarrow col(p.dest) - col(id),\ row(p.dest) - row(id)$
11: $target \leftarrow NULL$
12: **if** $\Delta x < 0$ **then** $target \leftarrow WEST$
13: **else if** $\Delta x > 0$ **then** $target \leftarrow EAST$
14: **else if** $\Delta y < 0$ **then** $target \leftarrow NORTH$
15: **else if** $\Delta y > 0$ **then** $target \leftarrow SOUTH$
16: **end if**

17: $id_{neighbor} \leftarrow getNeighbor(id, target)$

18: **if** $\neg isFull(noc[id_{neighbor}].buffer)$ **then** ▷ Synchronizing action goes here
19: $noc[id_{neighbor}].buffer \leftarrow enqueue(p)$
20: $b \leftarrow dequeue(b)$
21: $noc[id].thisActivityTemp \leftarrow noc[id].thisActivityTemp + 1$
22: $markAsServiced(b)$
23: **end if**
24: **end procedure**

A primary objective of this model is to quantify PSN. We track packet activity via the router-local `thisActivity` and `lastActivity` state variables, as opposed to a global PSN characterization present in [7]. The motivation for switching to a per-router PSN characterization and the reformulated PCTL properties are covered in Section 5.1. This section covers the implementation of the router process that ensures that stable, correct activity counters are available for the PCTL properties. Direct implementation of Properties 5.1 and 5.2 on the activity state variables of each router introduced transient state artifacts that were not seen when performing a global PSN characterization. These transient artifacts created artificially high PSN and incorrect probability calculation. These errors occured because `thisActivity` is calculated incrementally during routing, the model checker will evaluate the property on intermediate states (e.g., while the counter is rising but hasn't reached its final value for the cycle) as described in Section 5.1.

To ensure the property is evaluated only on the *stable* state at the end of a cycle when `thisActivity` is fully calculated, we implement a two-phase update with a temporary variable. We introduce a temporary variable, `thisActivityTemp`, which increments during the `AdvanceRouter` process. The main activity state variables are then updated in ordered steps around the clock action:

1. **Pre-Clock Edge:** `thisActivity` is assigned to the accumulated `thisActivityTemp` for the current clock cycle, and `thisActivityTemp` is reset.
2. **Clock Edge:** The global clock advances and PCTL properties can be correctly calculated.
3. **Post-Clock Edge:** The current activity from the previous clock cycle is stored into `lastActivity` to preserve the history for the next cycle.

Code Segment 4.21 shows the implementation of this noise tracking along with using synchronizing actions to reduce the state-space.

Code Segment 4.21. Two-Phase Activity Update

```
action thisActivityUpdate, lastActivityUpdate;

process UpdateThisActivity(int id) {
   thisActivityUpdate {=
      noc[id].thisActivity = noc[id].thisActivityTemp,
      noc[id].thisActivityTemp = 0
   =}
}

process UpdateLastActivity(int id) {
   lastActivityUpdate {=
      noc[id].lastActivity = noc[id].thisActivity
   =}
}
```

4.8 Abstract NoC Model

Despite the benefits provided by the synchronizing actions, model checking is still expensive for the MODEST model, and as the size of the NoC under verification grows, so does the verification cost for CTL properties both in time and statespace. A common technique to combat state-space explosion is *abstraction*, where parts of the model irrelevant to the property being verified are simplified or removed entirely. Because the modular NoC model is built from a composition of `Router` processes, verifying correctness of a single router process would provide high confidence that any size NoC is correct, as each of it's components (routers) were separately verified at an abstract level. To accomplish this abstracted verification, we model the *interface* between the router under verification and its immediate neighbors.

We apply abstraction by creating an abstract NoC model that focuses on a single router to be verified. In this thesis we model the central router of a 3×3 mesh, R_4. We chose to use a central router because each of its four possible neighboring connections are utilized. In our abstract, shown in Code Segment 4.22, the central router R_4 is a complete `Router` process, while its four cardinal neighbors are replaced with a abtract `AbstractNeighbor` process. The corner routers are removed entirely, as they do not directly interact with R_4,

and thus are not useful for the abstacted verification of R_4.

Code Segment 4.22. Abstract NoC Composition

```
par {
   :: Clock()
   :: Router(4) // Concrete Router
   :: AbstractNeighbor(1) // Abstracted North Neighbor
   :: AbstractNeighbor(3) // Abstracted West Neighbor
   :: AbstractNeighbor(5) // Abstracted East Neighbor
   :: AbstractNeighbor(7) // Abstracted South Neighbor
}
```

The `AbstractNeighbor` process (Code Segment 4.23) is the core of this abstraction. It simulates a valid, non-deterministic, environment for a router. It achieves this through two key simplifications over the `Router`: non-deterministic stimulus generation and non-deterministic backpressure.

First, instead of implementing a full traffic generation model, an abstract neighbor non-deterministically decides whether to inject a packet in each cycle. If it does, the packet is always destined for the router under test (`boundedEnqueue(4, ...)`), providing a constant source of incoming traffic through the central router.

Second, an abstract neighbor simulates backpressure from the rest of the (abstracted) network non-deterministically. The statement `isFull = DiscreteUniform(0, 1) == 1` randomly sets the status of the neighbor's buffer that receives packets from the router under test. This replaces the need to model the full state and routing decisions of the entire network beyond the neighbor, while still presenting a valid, dynamic environment where output channels might be blocked.

To handle packet transfers at the abstraction boundary, two specialized send processes are used. The `NeighborSend` process is used by an `AbstractNeighbor` to enqueue a real packet into a buffer of the concrete `Router` R_4. Conversely, when the concrete `Router` sends a packet to an abstract neighbor, it uses an `AbstractSend` process that simply dequeues the packet from the source buffer without enqueuing it anywhere, as its journey is complete from the perspective of the verification goal. This abstraction ensures that real packets

are concretely sent to the router under test, R_4, while packets leaving R_4 effectively are removed from the statespace.

Synchronization is maintained by ensuring the `AbstractNeighbor` process strictly adheres to the global synchronization steps described in Section 4.5. As shown in Code Segment 4.23, the abstract process executes the exact same sequence of synchronizing actions (`generateFlits`, `prepRouter`, `advanceChannel`, etc.) as a full `Router` process. This lockstep execution guarantees that from the perspective of the concrete router, its neighbors behave like valid, cycle-accurate components of the NoC, even though their internal logic is abstracted. This allows for the verification of single-router properties in isolation without altering its observed temporal behavior, which reduces the state space required for model checking.

Code Segment 4.23. The AbstractNeighbor Process

```
process AbstractNeighbor(int id) {
   // 1. Non-deterministic stimulus generation
   alt {
   :: generateFlits {= noc[id].channels[LOCAL].buffer =
            boundedEnqueue(4, noc[id].channels[LOCAL].buffer) =}
   :: generateFlits
   };

   // 2. Non-deterministic backpressure modeling
   prepRouter {=
      noc[id].channels[getNeighborChannel(id)].isFull = DiscreteUniform
         ↪ (0, 1) == 1,
      // ... prep local channel ...
   =};

   // 3. Simplified, uni-directional sending logic
   if (!noc[id].channels[LOCAL].isEmpty) {
      NeighborSend(id, LOCAL, getNeighborChannel(id))
   } else {
      advanceChannel; advanceChannelSend;
   };

   // 4. Strict synchronization adherence
   advanceChannel; advanceChannelSend;
   advanceChannel; advanceChannelSend;
   advanceChannel; advanceChannelSend;
   advanceChannel; advanceChannelSend;
   updatePriority;
   // ... more updatePriority actions ...
   updatePriority;
   nextClockCycle;

   AbstractNeighbor(id)
}
```

Checking the functional correctness of this abstract model provides high confidence that the `Router` process is correct for any size NoC.

4.9 Experiment Infrastructure and Reproducibility

The majority of this chapter has covered the design of the MODEST NoC model. This

section covers the efforts made to ensure that running the model is scientifically reproducible. To guarantee that the results presented in this thesis can be independently regenerated and verified, we produced an infrastructure for reproducible MODEST runs. This infrastructure focuses on two objectives: (1) environment isolation to prevent dependency failures, and (2) automated data capture to store the exact state of the system during every simulation. Fulfilling these two objectives render the results easily reproducible.

4.9.1 Environment Isolation

Each simulation or model checking task is managed by a Python script using a custom library that provides tooling for running MODEST and analyzing the models output.The project's dependency tree is saved using the standard `pyproject.toml` specification. This is a standard method for providing reproducible code in Python. Additionally, when each simulation or model checking task is completed the version of the MODEST TOOLSET used is saved along with the results from the tool.

4.9.2 Data Serialization

A challenge in formal verification experiments with many models is associating a result with the specific experiment that produced it. This is a challenge for our modular NoC model, because when generating results we often do a sweep over different paramters such as network size, packet injection duty cycle, or buffer length. Sweeping over these values creates many different models, and keeping track of them is a challenge. To address this, we created an representation of an experiment, `SimulationRun`, which serves as a schema the experiment data. This is implemented in Python using the `pydantic` library for strict typing, and stores four main pieces of data:

- **Inputs:** The exact NoC parameters set (e.g., mesh size, injection rate) used to generate the Modest model.
- **Model source:** A verbatim copy of the generated `.modest` code.
- **Execution context:** The raw CLI MODEST command, tool version, and timestamp.

- **Outputs:** The raw, unaltered stdout/stderr streams from MODEST, alongside parsed properties, and timing information.

4.9.3 Artifact Persistance

Upon completion of a simulation run or model checking task, the Python tools serialize the `SimulationRun` object into a standardized directory hierarchy. This system generates immutable[1], timestamped artifacts.

Each experiment produces a dedicated directory containing three important files:

1. `metadata.json`: A `.json` serialization of the experiment object, enabling automated post-processing and plotting.
2. `model_input.modest`: The exact source code fed to MODEST. This allows future researchers to re-run the specific case.
3. `modest_raw_output.txt`: The capture of MODEST's raw output, preserving warnings, errors, and properties.

This data-centric approach ensures that every data point presented in Chapter 6.8.1 is backed by the complete experiment chain, from the specific input model, the version of MODEST, and the complete output from MODEST. An artifact containing the research of this thesis is available on Zenodo [69].

[1] Technically, these are not truly *immutable*, as they are simply `.json` files. However, the tool treats them as immutable and will not overwrite them with future experiment data.

CHAPTER 5

Verification of MODEST NoC Models

This chapter details the verification methodology applied to the modular MODEST NoC model. It first addresses the refinements to PSN characterization, specifically the transition from global to local PSN metrics, and then outlines the application of CTL model checking for functional verification.

5.1 PSN Characterization via Probabilistic Verification

PSN quantification is accomplished by probabilistically bounding the network activity of routers over a fixed number of clock cycles. Previous probabilistic NoC models [6–8] characterized global PSN, this work building on the router-centric verification developed in [7] to provide higher fidelity PSN analysis, especially for larger networks.

5.1.1 Moving from Global to Per-router PSN Characterization

In past works, PSN is characterized by tracking the aggregate *activity* of the entire network via counting the number of *PSN events*. A PSN event occurs when a specific router has an activity (or change in activity) above a certain threshold. This method, established in [6,8] and expanded in [7], leverages the correlation between switching activity and current draw in the network [4]. These works produced *Cumulative Distribution Functions* (CDFs) representing the probability that the *total* system activity exceeds a threshold K within n cycles.

This aggregate metric offers distinct advantages for high-level analysis:

1. It produces a scalar metric allowing direct comparison between heterogenous architectures.
2. It requires tracking only two global state variables in addition to the NoC state and only two PCTL properties per clock cycle, which keep the state space and verification

time low, respectively.

However, global PSN metrics obscure the spatial (per-router) distribution of noise. A global measure cannot distinguish between a widely distributed network load and a localized "hotspot," yet this distinction is critical during physical design of a chip. Chip designers identify specific spatial regions with expected high current draw to optimize the power delivery network. Furthermore, while global metrics confirm the expected conclusion that "larger NoCs generate more PSN," they do not provide actionable insights for NoC optimization.

[7] partially addressed this issue with the introduction of PSN characterization per router. However, this method still used PSN events for characterization to maintain similarity with the global characterization. Using PSN events as a measure effectively adds an additional parameter for verification, as the *activity level* must be specified along with the event threshold K. This additional parameter muddies the meaning of per-router PSN metrics.

To address these limitations, we quantify PSN *per-router* using only the pre-existing activity counters in each router's state. By formulating properties that reason directly about `thisActivity` and `lastActivity` for each routers, we can quantify the probability of specific nodes exceeding resistive or inductive noise thresholds. This removes the additional parameter and counters needed the PSN event based analysis. One trade-off for this approach is verification time due the number of properties, i.e., verifying router-centric properties requires checking n^2 distinct properties per clock cycle (one per router) rather than a single global property.

5.1.2 Implementation Challenges and Resolution

Implementing router-centric verification requires checking that the activity of a specific router R_i eventually exceeds a threshold A within a bounded time horizon.

$$p_r = P_{=?}[\Diamond^{[\text{accumulate}(clk)\leq N]}\ R_i.thisActivity \geq A] \tag{5.1}$$

$$p_i = P_{=?}[\Diamond^{[\text{accumulate}(clk)\leq N]}\ |R_i.thisActivity - R_i.lastActivity| \geq A] \tag{5.2}$$

In MODEST, p_r and p_i are expressed as

```
property pr = Pmax(<>[S(clk)⩽n] noc[id].thisActivity ⩾ ACTIVITY_THRESH),
property pi = Pmax(<>[S(clk)⩽n] abs(noc[id].thisActivity -
    noc[id].lastactivity) ⩾ ACTIVITY_THRESH)
```

The Transient State Anomaly

Initial execution of this property on a 3×3 NoC using SMC yielded counter-intuitive results. While resistive noise probabilities were consistent with expectations from previous works, inductive noise estimates were abnormally high, approaching probability 1.0 at a higher-than-expected rate.

Analysis of the model revealed this to be a synchronization artifact, not true PSN. This synchronization artifact negatively impacted the inductive noise property (Equation 5.2) because it checks the difference between the current and previous activity. In PCTL, the eventually operator ($\Diamond$) evaluates the predicate at *every* state transition, however, our properties are written as if they were only evaluated on clock transitions. Our initial implementation reset `thisActivity` to zero at the start of each cycle and incremented it during routing. Consequently, immediately after the reset, the model entered a transient state where $R_i.thisActivity = 0$ while $R_i.lastActivity$ retained the previous cycle's high value (e.g., 3). This created a momentary, artificial difference:

$$\Delta_{transient} = |0 - 3| = 3 \geq A.$$

As this difference should be caught by the PCTL property, MODEST correctly detected this transient violation, resulting in erroneously high inductive noise.

Table 5.1 illustrates an example of a failing trace. This table shows an execution trace of a single router. Column one shows the clock cycle of the current router. Column two shows actions corresponding to the synchronizing actions and router processes shown in Code Segment 4.15. Columns three and four show the routers two activity trackers, *lastActivity* and *thisActivity*, which represent the activity on the last clock cycle and current clock cycle, respectively. Column five shows the inductive noise predicate p (Equation 5.2). Column six, p_{clk}, is the inductive noise property only checked at each clock edge.

This table shows how p and p_{clk} because of the transient updates to *thisActivity*. On clock cycle $n+1$, p becomes true at the "Update trackers" step because *thisActivity* is reset to 0. This is incorrect, as *thisActivity* will be incremented according to the activity of each buffer this clock cycle. When we get to clock cycle $n+2$, we can see that actually there is not change in activity between the two clock cycles, therefore, p should not hold.

Table 5.1: Trace of Transient State Violation (Incorrect Implementation)

Cycle	Action	*lastActivity*	*thisActivity*	p	p_{clk}
n	Clock edge	1	2	false	false
	Update trackers	2	**0**	false	
	Send packet north	2	1	false	
	...	...	...	...	
	Send packet east	2	3	false	
$n+1$	Clock edge	2	3	false	false
	Update trackers	**3**	**0**	**true**	
	Send packet south	3	1	true	
	...	...	...	...	
$n+2$	Clock edge	3	3	true	false

The Two-Phase Fix

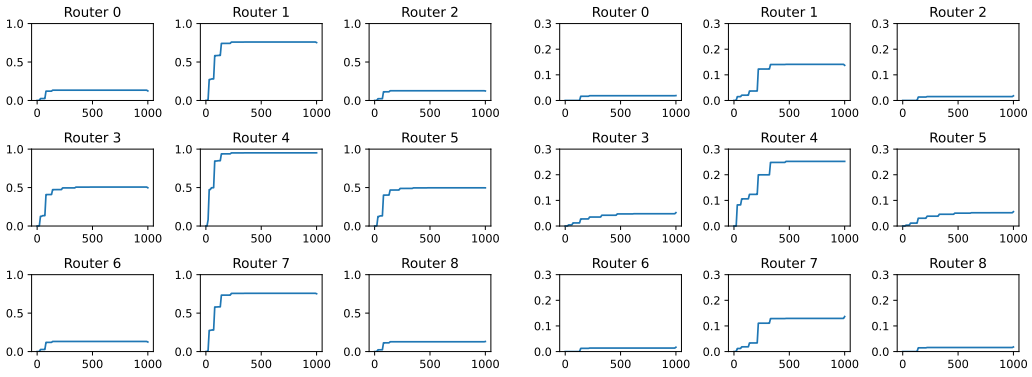
To resolve this, we implemented the **two-phase update mechanism** described in Section 4.7.4. We renamed the incremental counter to `thisActivityTemp` and introduced a stable state variable `thisActivity` that updates only at the clock action.

As shown in Table 5.2, this ensures that `thisActivity` remains constant throughout the cycle's sequential steps. The unstable values are contained within `activityTemp`, which is invisible to the property checker's target variables. Consequently, the predicate p is evaluated only on stable, valid states.

Table 5.2: Trace of Stable State Verification (Corrected Implementation)

Cycle	Action	*last*	*temp*	*this*	p	p_{clk}
n	Clock edge	1	2	2	false	false
	Update trackers	2	**0**	**2**	false	
	Send packet north	2	1	2	false	
	...	...	...	...		
$n+1$	Clock edge	2	3	2	false	false
	Update trackers	3	0	3	false	

Figure 5.1 demonstrates the impact of this correction. In the incorrect implementation (Fig. 5.1a), the probability of excessive inductive noise approaches 1.0 within 250 cycles due to the transient artifacts. In the corrected model (Fig. 5.1b), the probability stabilizes at approximately 0.25 after 1000 cycles, reflecting the expected behavior of the system.



(a) Incorrect (Transient Artifacts) (b) Corrected (Stable State)

Fig. 5.1: Impact of Split-Phase Updates on Inductive Noise Probability

5.2 CTL Verification of 2×2 NoC Models

This section details the methodology for the functional verification of the 2×2 modular

NoC. We employ the `mcsta` model checker from the MODEST TOOLSET [23] to validate the NoC. By leveraging the optimized modular framework to generate the complete state-space without a clock cycle bound, we verify $\forall\Box$ and $\exists\Diamond$ properties that were previously verified only through PCTL up to a bounded number of clock cycles in [7].

CTL verification of functional correctness provides additional certainty over the previously PCTL-based approach. Functional correctness properties formulated in PCTL typically follow a basic outline,

$$P_{pctl} = P_{=?}[\Diamond\Phi] = 1.0,$$

which states that the probability that property Φ eventually holds is 1.0. This is similar to the CTL property

$$P_{ctl} = \forall\Diamond\Phi,$$

which states that eventually Φ holds. However, P_{pctl} provides only an *almost-sure* guarantee that eventually Φ will hold, as there may be paths that have a 0.0 probability of occurring that violate the property. These 0.0 probability paths occur when an infinite execution of transitions with probability < 1.0 are chosen. While unlikely, these paths are still possible. Because of the introduction of CTL into MODEST recently, we will use CTL properties to check the functional correctness of the updated NoC model as CTL model checking does not suffer from almost-sure guarantees.

In the following definitions, we use the notation $p_j \rightarrow R_i.B_b$ to denote the event where a packet p destined for router R_j is enqueued in buffer b of router R_i. Additionally, MODEST does not have a quantification operator, so for properties quantified over all routers (e.g., $\forall i$) we manually unroll i into individual properties.

5.2.1 Verifying Traffic Injection Model

Since the modular framework supports interchangeable traffic injection models (Section 4.6), verification strategies must be adapted to the specific injection model. However, certain validity constraints are universal.

A fundamental safety requirement is that no PE may inject a packet destined for itself. We formalize this invariant in Property 5.3:

$$\forall i : \forall\Box(\neg(p_i \rightarrow R_i.B_{local})) \tag{5.3}$$

For specific patterns, such as the uniform distribution algorithm used in [8], we verify some additional properties. Specifically, we assert that for any pair of routers (R_i, R_j), there exists a valid execution path where R_i generates a packet destined for R_j. This ensures the injection logic is capable of stimulating the entire network:

$$\forall i : \forall j : \; j \neq i \implies \exists\Diamond(p_j \rightarrow R_i.B_{local}) \tag{5.4}$$

5.2.2 Verifying Arbitration Logic

Functional correctness of the Round-Robin arbiter relies on the integrity of the priority queue. The priority list must strictly represent a permutation of the five cardinal directions without duplication or omission at every state of execution. Property 5.5 enforces this structural invariant:

$$\begin{aligned} \forall i : \forall\Box(&\mathit{local} \in R_i.\mathit{priorityList} \; \wedge \mathit{north} \in R_i.\mathit{priorityList} \\ &\wedge \mathit{east} \in R_i.\mathit{priorityList} \; \wedge \mathit{south} \in R_i.\mathit{priorityList} \\ &\wedge \mathit{west} \in R_i.\mathit{priorityList}) \wedge \text{length}(\mathit{priorityList}) = 5 \end{aligned} \tag{5.5}$$

5.2.3 Verifying Flit Propagation

While the MODEST model uses dynamic list structures to model buffers, physical NoC hardware is constrained by fixed buffer depths. Additionally, in a physical NoC channel bandwidths are constrained and each channel can only be used once per clock cycle. We check these physical constraints are satisfied via safety properties.

Property 5.6 asserts that no buffer j in router i ever exceeds the configured maximum *BUFFER_SIZE*. Verification of this property confirms that the flow-control logic correctly

asserts backpressure to prevent overflow.

$$\forall i, j : \forall \Box(\text{length}(R_i.buffer_j) \leqslant BUFFER\_SIZE) \tag{5.6}$$

Additionally, to strictly enforce single-cycle channel semantics, we verify that no channel sends more than one packet per clock cycle. By instrumenting the model with a usage counter that resets every cycle, we confirm the absence of write-conflicts:

$$\forall i, j : \forall \Box(R_i.channel_j.send\_count \leqslant 1) \tag{5.7}$$

5.2.4 Optimization and State-Space Reduction

Before performing CTL model checking on the 2×2 NoC, we applied two optimizations. First, the synchronization barriers introduced in Section 4.5 reduce the state-space by eliminating spurious interleavings.

Second, we performed variable pruning by removing all PSN instrumentation from the model (only for functional verification). The variables `thisActivity`, `lastActivity`, and `thisActivityTemp` are *observer variables*, i.e., they track system behavior but do not influence control flow or routing. Including them in the CTL model forces the model checker to explore distinct states for every unique noise value, even when the underlying hardware state (buffers and channels) is identical.

By de-instrumenting the model, we separate the functional verification from the performance characterization. This abstraction significantly reduces the state space, rendering unbounded CTL verification tractable for the 2×2 design and establishing a foundation for future abstraction techniques on larger topologies.

5.2.5 Verification on the Abstract NoC Model

The abstract NoC model in Section 4.8 was developed to provide a strong guarantee of functional correctness of any $n \times n$ NoC model by demonstrating correctness for a generic router. The CTL properties from this section were applied to the instantiated router, R_4,

to accomplish this checking.

CHAPTER 6

Dafny Verification NoC Models

This chapter details the implementation of the square-mesh NoC model in the Dafny programming language and the methods used to match the execution semantics of the MODEST NoC model (Chapter 4). Translation of the functional correctness properties into Dafny is also shown.

6.1 Motivation and Scope

While the Modest-based verification in Chapter 5 successfully quantified PSN for large topologies (up to 12×12) using SMC and functional correctness for small topologies (up to 2×2) using PMC, it encountered the fundamental scalability barrier of model checking: *state-space explosion*. As the network dimension n increases, the state space grows exponentially, rendering explicit state enumeration and CTL model checking for functional correctness intractable for larger n under current memory constraints.

One approach to verifying functional correctness of arbitrary $n \times n$ NoCs was the abstract MODEST model presented in Section 4.8. However, because this abstract model still relies on model checking, only a single abstract router is checked, not an arbitrary network. An alternative approach to verifying an $n \times n$ model is deductive verification. Unlike model checking, which explores concrete states, deductive verification uses symbolic reasoning, mathematical induction, and modular proofs. This allows us to verify the functional correctness of Section 5.2 for *arbitrary* $n \times n$ topologies, proving that invariants hold for any valid configuration regardless of size.

This chapter details the translation of the asynchronous MODEST model into a sequential Dafny program. This translation contains a deliberate trade-off: while we gain the ability to prove unbounded functional correctness, we sacrifice the stochastic abilities of MODEST. Dafny has no support for probabilistic reasoning, and only limited support

for randomization. Therefore, this chapter focuses exclusively on functional correctness, complementing the quantitative probabilistic analysis of the previous chapters.

6.2 Why Dafny?

There are many tools that can be used for deductive verification [32, 34], however, this work focuses on using Dafny. Dafny is a verification-aware programming language [36] that incorporates an *auto-active* program verifier. Dafny is solely designed for sequential program verification, and as such is not natively suited to verifying digital circuits. However, the powerful auto-active program verifier provides fast, scalable, and often automatic verification of proofs. This automatic verification of proofs is one reason for choosing the Dafny language. Another reason is that it appears that Dafny has not been used hardware verification before in the literature, and it would be interesting to see if it works.

6.3 Translation Methodology

The initial starting point of the deductive NoC model is a translation from the MODEST model into Dafny. The primary objective of this translation is to preserve the architectural and execution semantics of the MODEST model to ensure that the verified properties apply to the original design. The following sections detail the engineering challenges inherent in adapting an asynchronous process algebra model to a sequential programming language. We specifically address the mapping of parallel composition to sequential method calls, the definition of inductive data structures, and the formulation of global invariants to prove functional correctness.

6.4 Modeling Paradigms and Architectural Choices

Translating the MODEST model into Dafny requires translating between two fundamentally different verification paradigms. MODEST uses a process algebra approach [70] to generate a concrete state space, with the addition of probabilities notated on transitions. In contrast, Dafny employs a deductive verification engine based on Floyd-Hoare logic [11, 12],

treating the program as a set of logical assertions (preconditions, postconditions, and invariants) to be proven symbolically.

This semantic gap requires a careful selection of the modeling architecture when translating from MODEST to Dafny. Leino [65] proposes two primary patterns for modeling concurrent systems (such as a NoC) in Dafny:

1. **Class-Based (Imperative):** Processes are encapsulated as objects with mutable state. Transitions are modeled as atomic class methods that modify this state. This approach is intuitive for hierarchical hardware modeling but requires explicit management of memory aliasing for each object. Additionally, this approach does not provide a method of encoding *time*, meaning that many liveness properties cannot be proven.
2. **State-Based (Functional):** The entire system state is modeled as an immutable data structure. Transitions are pure functions that map a state s to its next state s'. This facilitates liveness proofs but greatly complicates the modeling of localized updates as the NoC state is stored as a single object.

6.4.1 Evaluation of Modeling Strategies

This section covers the attempts at building the concurrent NoC system in Dafny.

Attempt 1: Class-Based Modeling

Our initial attempt mirrored the MODEST NoC data-structure directly, defining separate classes for `FlaggedBuffer`, `Buffer`, and `Router`. While structurally modular, this approach introduced significant complexity regarding dynamic frames (Preliminary Section 2.7.2).

In Dafny, a method must explicitly declare all memory locations it modifies. In a deep hierarchy, modifying a `Buffer` implies modification of its containing `FlaggedBuffer`, which implies modification of the `Router`. Validating a single packet injection required a cascading number of `modifies` clauses and "freshness" assertions (assertions that object references are not aliased) to prove to the verifier that distinct components did not alias the same memory.

The framing annotation overhead for these aliasing obligations grew with the model depth, rendering this approach unappealing.

Attempt 2: Functional State Modeling

Alternatively, we evaluated the functional approach: modeling the NoC as an immutable sequence of immutable Router states. A clock cycle was defined as a function transforming sequence $\mathcal{R}_t$ to $\mathcal{R}_{t+1}$, which entails transforming each router $r \in \mathcal{R}$ from state s to s'.

This approach was not further pursued due to the tight-coupling of NoCs. Unlike the loosely coupled ticket system in [65], a NoC requires n^2 routers to update *synchronously* based on the state of their neighbors. Expressing this global transition required constructing a new sequence of routers where every element is defined by a complex lambda expression involving neighbor lookups (Code Segment 6.1). The complexity of implementing a sending function that tracks whether each channel has already been used by another element grew to be deeply complex and unmanageable.

Code Segment 6.1. Complexity of Immutable State Updates

```
// Calculating the next state of the NoC requires a complex map
var next_routers := seq(|routers|, i ⇒
   if needs_update(routers[i]) then
      // Logic depends on neighbors, which are indices in the old seq
      // and whether or not a channel has already been used in the
      // current seq
      update(routers[i], routers[get_neighbor(i)])
   else routers[i]
);
```

Additionally, the resulting verification conditions involved deeply nested quantifiers over the state update functions. These quantifiers challenged the underlying SMT solver of the Dafny verifier, leading to unpredictable timeouts even for the smaller, unfinished test models.

Selected Approach: Flat Class Structure

To balance the tradeoffs of these two approaches, we selected a hybrid approach using a *flattened class structure*. We model the `Router` as a mutable class but eliminated several of the sub-classes in favor of native Dafny datatypes. This helped reduce the framing problem (by reducing hierarchy depth and the number of allocations) while avoiding the complexity of purely functional state updates. This function does limit the ability to prove liveness properties, however, as it does not easily encode time like the functional approach does because you cannot easily encode an infinite execution trace. In the functional case, you can model a trace of states as s, s', s'', ... with a function representing the updates between states. In the class structure though, s and s' are the same object, which limits the ability for Dafny to reason about the temporal changes in state. The MODEST model we are translating from does not verify any liveness properties of the form $\forall\Diamond$, so selecting a model in Dafny that does not easily allow for properties of that form meets our needs.

6.5 Router Components

To ensure semantic equivalence with the MODEST router state model (Chapter 4), the Dafny implementation constructs the router state using a composition of strongly-typed components. This section details the fundamental data types (e.g., packets, buffers, and directional wrappers) required to construct the router state model, highlighting the changes made to make the model more friendly for deductive verification.

6.5.1 Packet Representation

Consistent with Definition 2.1, a packet is abstracted as a integer value representing its destination indentifier. While in MODEST we model this as a bounded integer $p \in [0, n^2)$, in Dafny we model packets as unbounded natural numbers (`nat`). This abstraction allows the verification logic to remain separate from the specific mesh dimension n. Bounds checks are instead moved to the higher-level invariants.

Code Segment 6.2. Packet Type

```
type packet = nat
```

6.5.2 Buffer Implementation: Sequences vs. Recursive Types

In the MODEST model, buffers are implemented as recursive linked lists to support dynamic resizing. While Dafny supports similar recursive data types, we instead utilize the native sequence (`seq`) type to model buffers. A proof of equivalence between linked lists and sequences for our set of operations is given in Appendix A.

Rationale for Sequences

Although a linked-list implementation (shown below) offers a direct structural mapping to the MODEST model, it incurs a significant verification penalty. Dafny's underlying SMT solver possesses extensive built-in theorems and axioms for handling sequences. This allows Dafny to automatically prove properties about sequence length, concatenation, and slicing without manual proofs. In contrast, custom recursive types often require manual inductive proofs for basic operations. Additionally, operations such as concatenation and slicing are available with built-in Dafny syntax, while custom recursive types would require a library of functions to support these operations.

Code Segment 6.3. MODEST Buffer (Recursive)

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

Code Segment 6.4. Dafny Buffer (Recursive)

```
datatype buffer =
  | hd(id: int, tl: buffer)
  | tl
```

In Dafny we can use sequences as variables. While sequences are immutable, a sequence variable may be updated by assignment to an entirely new sequence. This allows us to easily map hardware buffer state updates (e.g., enqueue, dequeue, peek) onto sequence variables.

Buffer Semantics

Definition 2.2 specifies the four required buffer operations. As demonstrated in Appendix A.1, Dafny sequences are semantically equivalent to linked lists for these operations but offer superior syntax and automated proving.

Table 6.1: Semantic Mapping of Buffer Operations

Operation	**Modest (List)**	**Dafny (Sequence)**
Enqueue	`enqueue(n, b)`	`b + [n]`
Dequeue	`dequeue(b)`	`b[1..]`
Peek	`peekFront(b)`	`b[0]`
Length	`len(b)`	`|b|`
Containment	`contains(n, b)`	`n in b`

To support the synchronous flagged behavior of the MODEST model (where status flags like `isFull` persists even if the buffer content changes transiently), we encapsulate the sequence in a `Buffer` datatype (Code Segment 6.5). This wrapper exposes helper functions that separate data manipulation from flag updates, essential for proving non-interference properties during the `PrepRouter` phase.

Code Segment 6.5. Buffer Datatype Wrapper

```
datatype Buffer =
| Buffer(buffer: seq<nat>, isEmpty: bool, isFull: bool) {
  static function init(): Buffer {
    Buffer([], true, false)
  }

  /* ... Helper accessors omitted for brevity ... */

  function insert(id: nat): Buffer {
    Buffer(this.buffer + [id], this.isEmpty, this.isFull)
  }

  // Separation of concerns: Flags are updated independently of content
  function setFlags(buffer_length: nat): Buffer {
    Buffer(this.buffer, this.length() == 0, this.length() ≥
        ↪ buffer_length)
  }

  function peekFirst(): nat
    requires this.length() > 0 // AG Reasoning: Precondition enforced by
        ↪ solver
  {
    this.buffer[0]
  }
}
```

Note the use of the `requires` clause in `peekFirst`. Unlike the MODEST model, where an invalid peek might return a "invalid" value (e.g., -1) or crash at runtime, Dafny statically enforces that `peekFirst` can never be called on an empty buffer. If the verifier cannot prove this precondition at the call site, verification fails at compile time.

6.5.3 The Direction Wrapper Container

In the MODEST model, each buffer is stored in an array indexed by integers (e.g., `buffers[0]` to access the North buffer). While MODEST relies on state-space enumeration to catch indexing errors (e.g., accessing `buffers[5]` is invalid), deductive verification places the burden of proof on the caller. Using raw arrays to store each of our five input buffers in Dafny would require every method to carry preconditions proving that indices are within bounds

($0 \leq i < 5$) and to ensure that the buffers are always at the correct index. Additionally, any access to a buffer through the array would lose part of its meaning, as accessing `buffer[i]` would not explicitly provide information about which direction is being accessed.

To eliminate this verification overhead and provide greater clarity, we introduce the `DirectionWrapper<T>`, a generic class that replaces array indexing with named structural fields (`north`, `east`, etc.). This approach trades iterability for type safety.

- **Benefit:** Accessing `buffers.north` is structurally safe (no bounds check is required) and clearly indicates its meaning.
- **Drawback:** Accessing a specific direction via a variable requires using a `match` or `if` statement.

To bridge this gap, the class provides `fromDir` and `writeByDir` methods (Code Segment 6.6), allowing indexed access by variable when necessary while preserving the safety of named fields.

Code Segment 6.6. Direction Wrapper Class

```
class DirectionWrapper<T> {
  var north: T; var east: T; var south: T; var west: T; var local: T

  /* ... Constructor omitted ... */

  // Facilitates iteration for verification properties
  function asSeq(): (s: seq<T>)
    reads this
    ensures |s| == 5
  {
    [this.north, this.east, this.south, this.west, this.local]
  }

  // Maps enumerated direction to structural field
  function fromDir(dir: Direction): T
    reads this`north, this`east, this`south, this`west, this`local
  {
    match dir
    case North ⇒ this.north
    case East ⇒ this.east
    // ...
```

```
  }

  // Verified update logic
  method writeByDir(dir: Direction, data: T)
    modifies this`north, this`east, this`south, this`west, this`local
    // Frame condition: Only the target field changes. This informs
    // the verifier that only the data corresponding to `dir` updates
    // and all other fields remain unchanged. This helps limit the
    // verification task, and also ensures verification integrity
    // across calls to `writebyDir`.
    ensures dir == North ==> north == data && unchanged(`east) && ...
  {
    match dir
    case North => { this.north := data; }
    case East => { this.east := data; }
    // ...
  }
}
```

6.6 Router Class Implementation

Having defined the required data types, we now detail the implementation of the `Router` class. This section describes the following methods for the router class: construction, packet generation, flag management, packet transmission, and arbitration.

6.6.1 Class Definition and State

The `Router` class models the state required to implement the formalized router definition (Definition 2.3). In addition to the `Buffer` and `DirectionWrapper` types, we also use a custom type, `FixedSeq`, to enforce the structural invariant that the priority list always contains exactly five elements (Code Segment 6.7).

Code Segment 6.7. Fixed Sequence Type

```
type FixedSeq<T> = i: seq<T> | |i| == 5 witness *
```

The class state is divided into constant metadata (reflecting the static nature of real NoC hardware) and mutable runtime state (Code Segment 6.8). Additionally, there is

phantom (non-compiled) state used to assist in verification tasks.

Code Segment 6.8. Router State Variables

```
// Static metadata
const id: nat
const dim: nat
const buffer_length: nat

// Input/Output State
const buffers: DirectionWrapper<Buffer> // 𝓑 in Def. 2.3
const serviced: DirectionWrapper<bool>
const used: DirectionWrapper<bool> // 𝓒 in Def. 2.3

// Arbitration state (𝓐 in Def. 2.3)
var priority_list: FixedSeq<Direction>

// Phantom verification state
ghost var all_packets: set<nat>
```

Note that while `buffers`, `serviced`, and `used` are marked `const`, they are references to mutable `DirectionWrapper` objects. In this case, `const` means that the reference (pointer) will not change, but the underlying object's member variables may still change. This pattern allows us to leverage Dafny's `fresh` predicate in the constructor to prove that these components are uniquely allocated and non-aliased, satisfying the requirement that these variables must be unique between unique routers.

6.6.2 Constructor and Initialization

The constructor's (Code Segment 6.9) primary role is to establish the initial invariants of the `Router` class. It accepts the topological metadata as initialization parameters and guarantees strict validity constraints on the metadata via the `valid` predicate. The constraints that the dimension is valid ($n \geq 2$) and that the identifier lies within topological bounds ($id \in [0, n^2)$) are fundamental properties of the square mesh NoC architecture.

Critically, the constructor guarantees that each internal object (buffers, serviced flags, and used flags) is newly allocated. In Dafny's verification logic, the `fresh` predicate asserts that these newly allocated objects do not overlap with any previously allocated objects on

the heap. This guarantees heap isolation which ensures that each routers sub-objects are unique and do not alias an object in any other router.

Finally, the constructor ensures the system initializes to a deterministic, default state where all buffers are empty, flags are cleared, and the priority list is set to the default order.

Code Segment 6.9. Router Constructor

```
predicate validMetadata() {
  && this.dim ⩾ 2
  && 0 ⩽ this.id < (this.dim*this.dim)
  && buffer_length > 0
}

constructor(buffer_length: nat, id: nat, dim: nat)
  requires buffer_length > 0
  requires dim ⩾ 2
  requires 0 ⩽ id < dim*dim
  ensures this.id == id
  ensures this.dim == dim
  ensures this.buffer_length == buffer_length
  // Heap Isolation Guarantee: Wrappers are distinct and non-aliased
  ensures fresh(this.buffers) && fresh(this.serviced) && fresh(this.used
     ↪ )
  // Initialization State
  ensures this.buffers.north.length() == 0
  ensures this.buffers.east.length() == 0
  ensures this.buffers.south.length() == 0
  ensures this.buffers.west.length() == 0
  ensures this.buffers.local.length() == 0
  ensures this.serviced.asSeq() == [false, false, false, false, false]
  ensures this.used.asSeq() == [false, false, false, false, false]
  ensures this.priority_list == [North, East, South, West, Local]
  ensures validMetadata()
  ensures this.all_packets == {}
{
  this.id := id;
  this.dim := dim;
  this.buffer_length := buffer_length;
  this.buffers := new DirectionWrapper(Buffer.init(), Buffer.init(),
     ↪ Buffer.init(), Buffer.init(), Buffer.init());
  this.serviced := new DirectionWrapper(false, false, false, false,
     ↪ false);
  this.used := new DirectionWrapper(false, false, false, false, false);
```

```
  this.priority_list := [North, East, South, West, Local];
  this.all_packets := {};
}
```

6.6.3 Execution Semantics and Ordering

To maintain valid semantic equivalence, the Dafny `Router` class must replicate the execution order of the MODEST model. While the MODEST model uses a recursive process call to model each clock cycle, the Dafny model uses sequential method calls within an infinite loop (see Section 6.7.2).

Code Segment 6.10 shows the execution loop defined in the MODEST model. The Dafny implementation strictly follows this four-phase sequence, i.e., Generation, Preparation, Routing, and Arbitration. This ensures that, once verified, the functional correctness properties apply back to the original system architecture. The MODEST model additionally has a PSN instrumentation phase, but the Dafny model does not implement this due to the focus on functional correctness and Dafny's lack of randomization support.

Code Segment 6.10. Reference Execution Loop in MODEST

```
process Router(int id) {
   GenerateFlits(id);
   PrepRouter(id);
   AdvanceRouter(id);
   UpdatePriority(id);
   nextClockCycle;
   Router(id)  /* Recursive Call */
}
```

6.6.4 Traffic Injection (GenerateFlits)

The `generateFlits` method[1]is responsible for introducing new traffic into the network.

Initial Approach to Packet Generation

Our initial implementation (Code Segment 6.11) attempted to replicate the MODEST

[1]We retain the `generateFlits` name to maintain traceability to the MODEST model, despite the use of single-flit packets.

model's behavior directly within the router class. In MODEST, traffic generation is an internal stochastic process. Since Dafny is a deterministic verification language, we modeled this stochastic behavior non-deterministically using an unimplemented method, `getPackets`.

This method serves as a source of non-determinism, returning an arbitrary valid destination identifier. By marking the postcondition as an `axiom`, we instruct the verifier to assume a valid value exists without requiring an implementation, effectively modeling a random variable. This models our stochastic traffic generation process because any valid destination identifier may be generated every clock cycle.

Code Segment 6.11. Internal Packet Generation (Initial Attempt)

```
method generateFlits(cycle: nat) {
  // Injection Policy coupled with Mechanism
  if cycle % 3 < 3 && |this.buffers.local.buffer| < this.buffer_length {
    var dest := this.getPackets();
    this.buffers.local := this.buffers.local.insert(dest);
  }
}

method getDestination() returns (dest: nat)
  // Returns a valid non-self destination
  ensures {:axiom} 0 ⩽ id < dim*dim && dest ≠ id
```

Improved Approach to Packet Generation

While the initial approach to packet generation was functional, it coupled the traffic injection pattern (logic determining *when* and *what* to send) with the router (logic determining *how* to send). This deviates somewhat from the conceptual NoC architecture (Section 2.2), where an external PE drives the traffic injection.

Leveraging Dafny's AG features, we refactored the method to decouple traffic injection and router operation (Code Segment 6.12). The router now exposes a passive injection interface. The responsibility for checking buffer capacity and generating a valid packet is shifted to the caller (the PE or external environment) via preconditions.

This refactoring yields two significant verification benefits:

1. **Simplified safety proofs:** The precondition `|local| < buffer_length` allows Dafny to easily verify that the router never overflows its local buffer in this method.
2. **Flexible traffic injection:** The traffic pattern is no longer hard-coded in the router, allowing arbitrary injection models to be verified or implemented without modifying the router class.

The refactoring shown in Code Segment 6.12 takes an `Option` type (from the Dafny standard library), which is a datatype with two variants, `Some` and `None`. We check if the optional holds a value with `Some?`. `generateFlits` uses this optional to determine whether or not a packet will be added to the local buffer.

Code Segment 6.12. External Packet Injection

```
method generateFlits(dest: Wrappers.Option<nat>)
  requires validMetadata()
  // Assumption: Caller must respect backpressure
  requires dest.Some? ⟹ |this.buffers.local.buffer| < this.
      ↪ buffer_length
  // Assumption: Caller must provide valid packet
  requires dest.Some? ⟹ isValidId(dest.value) && dest.value ≠ this.id
  modifies this.buffers`local
  ensures |this.buffers.local.buffer| ⩽ this.buffer_length
{
  if dest.Some? {
    this.buffers.local := this.buffers.local.insert(dest.value);
  }
}
```

6.6.5 Buffer State Sampling (PrepRouter)

The `prepRouter` method corresponds to the preparation phase of the MODEST model (Section 4.3.2). Its purpose is to sample the observable status flags (`isEmpty`, `isFull`) with the contents of the buffers at the start of a cycle. In a hardware design, these flags would always update synchronously with each buffer. However, the NoC model in this work updates parts of the NoC serially, meaning that there are transient states between clock cycles where we require these flags to stay the same.

The postconditions in Code Segment 6.13 ensure that the flags are updated correctly (e.g., `isFull` is set iff the buffer is full) and that the data in each buffer remains unchanged.

1. **Data preservation:** `old(buffer) == buffer` ensures no packets are lost or re-ordered.
2. **Flag correctness:** The flags are proven to accurately reflect the buffer state relative to the `buffer_length` parameter.

Code Segment 6.13. Buffer Stabilization Method

```
method prepRouter(cycle: nat)
  requires validMetadata()
  modifies this.buffers

  // Data Preservation Contract: Packets must not change
  ensures old(this.buffers.north.buffer) == this.buffers.north.buffer
  // ... repeated for all directions ...

  // Flag Consistency Contract: Metadata matches state
  ensures (this.buffers.north.length() == 0 ⟺ this.buffers.north.
      ↪ isEmpty)
      && (this.buffers.north.length() ⩾ this.buffer_length ⟺ this.
          ↪ buffers.north.isFull)
  // ... repeated for all directions ...
{
  this.buffers.north := this.buffers.north.setFlags(this.buffer_length);
  // ... assignments for other directions ...
}
```

6.6.6 Sending Packets (AdvanceRouter)

The fundamental purpose of a NoC is packet transport. Therefore, the packet transmission logic is a critical path for verification. While the algorithmic logic of the Dafny NoC mirrors the MODEST model (servicing buffers in priority order), the architectural implementation requires significant adaptation to satisfy Dafny's heap-based deductive verification model.

In MODEST, the NoC is modeled as a global array of router processes. A router R_i sends a packet to R_j by directly modifying the state of index j in the global array. The

validity of this operation relies on the explicit state-space exploration of the model checking engine. In Dafny, however, each router is a distinct object on the heap. A router cannot simply "reach out" and modify another router object. Instead, modifying another router requires a valid reference and a proof that modifying that reference does not violate the invariants of the other router.

This problem will be referred to as the *neighbor connectivity problem*, as the challenge is developing a deductive method to prove the validity of R_i sending a packet to (and thus modifying) R_j. We explored three distinct implementations to solve this problem while working toward a solution that balanced implementation complexity with verification tractability.

Attempt 1: Neighbor References

Our initial approach attempted to provide neighbor connectivity by storing references to neighboring routers directly within the `Router` class (Code Segment 6.14).

Code Segment 6.14. Neighbors as Router Member Variables

```
class Router {
// ...
var registered: bool
var north_neighbor: Router?
// ... other directions ...

// Method to link neighbors after construction
method registerNeighbors(north: Router?, east: Router?, ...)
  requires !registered
  ensures north_neighbor == north // ...
  ensures validTopology()
{ ... }

// Invariant asserting topology validity
predicate validTopology() {
  && registered
  && north != null ==> north != this
  && east != null ==> east != this
  // ...
}
}
```

While this provided a convenient interface during packet transmission (e.g., `this.north_neighbor.enqueue( ... )`), it was challenging to verify the *framing* of these member variables. To verify any method that modifies another router reference, Dafny requires proof that the modification does not invalidate the invariants of other objects. By embedding references to other routers inside the state of the current router, we created a recursive dependency graph. Proving that `r1.send()` does not modify `r2` when `r2` is a neighbor of `r1` required maintaining global invariants about the entire graph structure in every method contract. This rendered the verification process intractable.

Attempt 2: External Neighbor Injection

To decouple the router state from the topology, we attempted to pass neighbors dynamically to the `advanceRouter` method (Code Segment 6.15). Instead of storing references, the caller would provide a collection of relevant neighbors for the current cycle.

This shifted the burden of topological correctness to the caller. The router no longer needed to maintain object invariants for the neighbors, it only needed to verify that the provided neighbors were valid for the `advanceRouter` method.

Code Segment 6.15. Neighbor Injection Interface

```
method advanceRouter(neighbors: DirectionWrapper<Wrappers.Option<Router
    ↪ »)
  requires validMetadata()
  // Quantified preconditions proving neighbor validity
  requires forall n | n in neighbors.asSeq() :: n.Some? ⟹ this.dim =
      ↪ n.value.dim
  requires forall n | n in neighbors.asSeq() :: n.Some? ⟹ this.id ≠ n
      ↪ .value.id
  requires forall n | n in neighbors.asSeq() :: n.Some? ⟹ n.value.
      ↪ validMetadata()
  modifies this.serviced, this.buffers
  modifies (set x | x in neighbors.asSeq() && x.Some? :: x.value.buffers
      ↪ )
{
  for i := 0 to |this.priority_list| {
    var dir := this.priority_list[i];
    var n := neighbors.fromDir(dir);
    if n.Some? && this.buffers.fromDir(dir).length() > 0 {
      this.advanceChannel(n.value, dir);
    }
  }
}
```

While this decoupled the neighbor references from the router class, the verification overhead remained high. The method signature required several quantified preconditions (`forall ...`) to establish the validity of the neighbor set. Furthermore, it separated the *logic* of neighbor calculation (handled by the caller) from the *usage* (handled by the router). This separation both made it difficult to prove properties that depended on specific topological relationships (e.g., "the North neighbor has a valid identifier that is $id - dim$"), and added verification overhead where the mathematical neighbor calculation had to be encoded a precondition. The combination of the quantifiers and neighbor calculation preconditions caused significant slowdowns in verification, and often resulted in timeouts (10+ minutes

on a M4 MacBook Pro) where no verification conclusion could be reached.

A timeout of 10 minutes may seem quite short, especially compared to the time required for large model checking problems. However, in Dafny verification typically takes less than a minute, even for highly complex problems. This is due to it's powerful AG approach to verification. When dealing with timeouts, it's often a good practice to focus on refining your verification approach, rather than giving the SMT solver more time. The more time you give the solver, the closer you approach a model checking-like solution where the SMT solver has to explore all possible states.

Solution: Global NoC Access

To resolve these issues, we adopted a pattern that mimics the MODEST implementation while remaining tractable for deductive verification within Dafny. Instead of passing specific neighbors, we pass the entire immutable sequence of routers representing the NoC to `advanceRouter` (Code Segment 6.16).

This approach allows each router to dynamically calculate neighbor indices (e.g., $id - dim$ for North) and access them directly from the sequence. This restores the simplicity of the arithmetic topology definition used in MODEST.

Additionally, the preconditions quantifying correctness over each router did not present any significant slowdowns for verification, with verification times remaining under 30 seconds.

An key part of this method is the three loop invariants. These invariants ensure

1. every router's metadata remains valid,
2. each loop iteration can modify the `serviced`, `used`, and `buffers` class members, and
3. that each `buffer` in `buffers` can be modified.

Without these loop invariants, the Dafny verification engine will assume that each router may be invalid, and the verification engine will not allow modification of any of the referenced class members.

Code Segment 6.16. Global NoC Access Implementation

```
method advanceRouter(routers: seq<Router>)
  requires validMetadata()
  requires |routers| == this.dim*this.dim
  requires routers[this.id] == this
  // Global validity invariant
  requires forall r | r in routers :: r.validMetadata() && r.dim == this
      ↪ .dim
  // Strict ID-to-Index mapping
  requires forall i | 0 <= i < |routers| :: routers[i].id == i

  // Frame condition: Only modify buffers in the system
  modifies this.serviced, this.used
  modifies set x | x in routers :: x.buffers
{
  for i := 0 to |this.priority_list|
    // Loop invariants
    invariant forall r | r in routers :: r.bufferLengthsvalidMetadata()
    modifies this.serviced, this.used, this.buffers
    modifies set x | x in routers :: x.buffers
  {
    match this.priority_list[i]
    case North => {
      var id_ := this.id - this.dim;
      // Arithmetic check replaces complex object graph proof
      if isValidNeighborId(id_) && this.buffers.fromDir(North).length() >
          ↪  0 {
        this.advanceChannel(routers[id_], North);
      }
    }
    case East => {
      var id_ := this.id + 1;
      if isValidNeighborId(id_) && this.x() < this.dim - 1 && this.
          ↪ buffers.fromDir(East).length() > 0 {
        this.advanceChannel(routers[id_], East);
      }
    }
    // ... South, West cases ...
    case Local => {}
  }
}
```

By passing the global sequence, we leverage Dafny's efficiency with sequences and

arithmetic. Proving that `routers[id - dim]` is a valid neighbor is computationally cheaper than proving `neighbors.fromDir(North)` satisfies a complex object invariant. This method successfully verified packet correctness and buffer safety where previous attempts timed out.

6.6.7 Arbitration and Fairness (UpdatePriority)

The `updatePriority` method is responsible for enforcing fair access to the output channels via a Round-Robin arbitration strategy. The goal is to permute the `priority_list` such that buffers that were serviced in the current cycle are moved to the lowest priority, while unserviced buffers are promoted to a higher priority.

The Naive Implementation Failure

Our initial attempt mirrored the MODEST implementation: iterating through the priority list and sorting elements into a temporary array based on their serviced status (Code Segment 6.17).

Code Segment 6.17. Naive Priority Shuffle

```
var priority_list_temp: FixedSeq<Direction>;
var serviced_index := 0; var unserviced_index := 0;

while i < 5 {
  if this.serviced(priority_list[i]) {
    var index := totalUnserviced + serviced_index;
    // Verification failure - index out of bounds
    priority_list_temp := priority_list_temp[index := priority_list[i]];
    serviced_index := serviced_index + 1;
  } else {
    var index := unserviced_index;
    priority_list_temp := priority_list_temp[index := priority_list[i]];
    unserviced_index := unserviced_index + 1;
  }
}
```

This implementation proved formally intractable. Dafny failed to verify array bounds safety because the solver could not deduce that the sum of two independent counters ($totalUnserviced + servicedIndex$) is bounded by the array length, even though their ranges

partition the space. Furthermore, proving that the resulting list was a valid permutation (containing all 5 directions exactly once) would require complex inductive invariants that related the in place creation of the temporary priority list to the validity of the original priority list.

Algorithmic Refinement: Stable Partitioning

To resolve this, we re-evaluated the algorithmic intent of `updatePriority`. The Round-Robin update is mathematically equivalent to a *stable partition* operation. We must partition each of the five directions $\{N, E, S, W, L\}$ in the priority list into two sequences, U (unserviced) and S (serviced), while preserving the relative order of elements within each set. Then we can construct the new priority list by concatenating U and S, ensuring that U comes before S. This stable partition ensures that the unserviced buffers will be at the front of the priority list following `updatePriority`.

We implemented this using an in-place rotation algorithm (Code Segment 6.18). We maintain a sequence of indices, initially $[0, 1, 2, 3, 4]$. We iterate through the sequence and when we encounter a "serviced" index at the frontier of the unserviced region, we rotate it to the back of the array.

The three loop invariants used in this algorithm ensure that the algorithm works correctly.

1. The `unserviced_index` is bound by `i`. This is required because the frontier of the unserviced region cannot exceed the iteration index.
2. Each of the indices must remain bounded between zero and four. This ensures that the indices cannot become invalid.
3. Each out the indices must be unique. When combined with 2., this means that the loop maintains a unique set of indices $\{0, 1, 2, 3, 4\}$.

Code Segment 6.18. Stable Partition via Rotation

```
// Get a sequence of current serviced status by the current priority
    ↪ order
var s_in_order := Seq.Map((x) reads this.serviced ⇒ this.serviced.
    ↪ fromDir(x), this.priority_list);

// Invariant: Indices are always a valid permutation of 0..4
for i := 0 to 5
  invariant 0 ⩽ unserviced_index ⩽ i
  invariant forall j: nat | j < 5 :: 0 ⩽ indices[j] < 5
  invariant forall j: nat, k: nat | j ≠ k && j < 5 && k < 5 :: indices[
      ↪ j] ≠ indices[k]
{
  if s_in_order[indices[unserviced_index]] {
    // Element is serviced. Rotate it to the end of the sequence.
    // [0, 1, 2, 3, 4] → [0, 2, 3, 4, 1] (if 1 was serviced)
    indices := indices[0..unserviced_index] + indices[unserviced_index
        ↪ +1..] + [indices[unserviced_index]];
  } else {
    // Element is unserviced. Leave it in place and advance frontier.
    unserviced_index := unserviced_index + 1;
  }
}
```

This approach simplifies the verification burden significantly. Instead of proving arithmetic bounds on computed indices, we need only prove that the rotation operation preserves the "permutation invariant" (i.e., the set of indices remains $\{0..4\}$). Dafny's built-in sequence theorems solve this automatically.

Finally, we construct the new priority list by mapping the permuted indices back to the original directions (Code Segment 6.19). The two loop invariants ensure that all of the indices are unique, and that all entries in the priority list are unique. These two invariants guarantee that after the for-loop `priority_list_temp` will be a permutation of `priority_list`.

Code Segment 6.19. Constructing the Next Priority List

```
for i := 0 to 5
  invariant forall j, k | j ≠ k && j < i && k < i :: indices[j] ≠
      ↪ indices[k]
  invariant forall j, k | j ≠ k && j < i && k < i :: priority_list_temp
      ↪ [j] ≠ priority_list_temp[k]
{
  priority_list_temp := priority_list_temp[i := this.priority_list[
      ↪ indices[i]]];
}
```

6.7 NoC Datatype Implementation

This section details the construction of an $n \times n$ NoC in the Dafny model, utilizing the router class defined in Section 6.6. We introduce the `construct` method, which establishes the structural invariants of the NoC, and the `run` method, which models the infinite execution of the system.

As defined in Definition 2.4, a NoC consists of a set of routers $\mathcal{R}$, a set of connections $\mathcal{C}$, and a dimension n. A standard implementation might define a collection type explicitly storing $\mathcal{R}$ and $\mathcal{C}$. However, since this work focuses exclusively on a square $n \times n$ mesh topology, $\mathcal{C}$ can be derived on-the-fly deterministically from the dimension n. Additionally, $\mathcal{R}$ can be constructed automatically with only n. Therefore, the NoC datatype (Code Segment 6.20) only needs to statically store the dimension n.

Code Segment 6.20. NoC Datatype Structure

```
datatype NoC = NoC(dim: nat)
```

Based on this definition, two primary methods manage the system lifecycle. The `construct` method operates on the `NoC` datatype to instantiate a set of routers $\mathcal{R}$. The connections between routers $\mathcal{C}$ is realized implicitly within the `run` method, which models the synchronous execution of the network. These methods collectively enable the verification of the NoC's functional correctness properties.

6.7.1 The Construct Method

The `construct` method, defined within the `NoC` datatype, instantiates the set of routers $\mathcal{R}$. As illustrated in Code Segment 6.21, this method requires that the network dimension satisfies $n \geq 2$ and that the specified buffer capacity is strictly positive. The post-conditions guarantee that the resulting sequence contains exactly n^2 routers, that all router instances are distinct objects (`allUnique`), and that every router satisfies the validity predicate defined in Code Segment 6.22.

The implementation utilizes a loop to allocate and append new `Router` instances. Loop invariants permit the verifier to inductively prove that the partial sequence maintains correctness at every iteration. By encapsulating this logic, we ensure the consistent initialization of $\mathcal{R}$ regardless of the network size.

Code Segment 6.21. Construct Method

```
method construct(buffer_length: nat) returns (routers: seq<Router>)
    requires dim ≥ 2
    requires buffer_length > 0
    ensures |routers| == dim*dim
    ensures allUnique(routers)
    ensures forall j | 0 ≤ j < dim*dim :: routers[j] is valid
  {
    routers := [];
    for i := 0 to dim*dim
      invariant 0 ≤ i ≤ dim*dim
      invariant |routers| == i
      invariant allUnique(routers)
      invariant forall j | 0 ≤ j < i :: routers[j] is valid
    {
      var r := new Router(buffer_length, i, this.dim);
      routers := routers + [r];
    }
  }
```

Establishing the validity of a router requires satisfying a conjunction of predicates (Code Segment 6.22). These clauses verify that the router and its reference-type fields are freshly allocated on the heap, preventing aliasing errors. Furthermore, the predicate asserts functional correctness of the initial state: buffers are empty, service flags are cleared,

and the priority list is initialized to the default Round-Robin order. A critical invariant enforced here is that a router's identifier must match its index within the $\mathcal{R}$ sequence, i.e., `routers[j].id == j`.

Code Segment 6.22. Initial State Validity Predicate for $\mathcal{R}$

```
&& fresh(routers[j])
&& fresh(routers[j].buffers)
&& fresh(routers[j].serviced)
&& fresh(routers[j].used)
&& routers[j].id == j
&& routers[j].dim == this.dim
&& routers[j].buffer_length == buffer_length
&& routers[j].buffers.north.length() == 0
&& routers[j].buffers.east.length() == 0
&& routers[j].buffers.south.length() == 0
&& routers[j].buffers.west.length() == 0
&& routers[j].buffers.local.length() == 0
&& routers[j].serviced.asSeq() == [false, false, false, false, false]
&& routers[j].totalUnserviced == 0
&& routers[j].used.asSeq() == [false, false, false, false, false]
&& routers[j].priority_list == [North, East, South, West, Local]
&& routers[j].validMetadata()
```

In the MODEST model, such initialization is static and fixed for a specific dimension. In contrast, the ability to specify this logic as a parameterized, formally verified method highlights the flexibility of the deductive approach.

6.7.2 The Run Method

The `run` method, defined within the `NoC` datatype, models the infinite execution of the network. This section details the method's implementation, explicitly mapping the execution semantics of the MODEST process algebra to the sequential Dafny environment. We further clarify how the set of connections $\mathcal{C}$ is realized implicitly through neighbor calculation logic.

Initialization Strategy

A primary design decision is whether to accept an existing set of routers $\mathcal{R}$ as an input

or to instantiate $\mathcal{R}$ within `run`. While semantically equivalent–assuming the input $\mathcal{R}$ satisfies all validity predicates–passing $\mathcal{R}$ as an argument requires the caller to satisfy the extensive initial state predicate (Code Segment 6.22). To encapsulate this complexity and ensure a correct initial state by construction, we call `construct` internally. Code Segment 6.23 illustrates this structure.

Code Segment 6.23. Run Method Skeleton

```
method run(buffer_length: nat)
  requires dim ⩾ 2
  requires buffer_length > 0
{
  var routers := this.construct(buffer_length);
  // ...
}
```

Modeling Infinite Execution

The Dafny implementation must emulate the behavior of the MODEST model (Code Segment 6.10). In MODEST, infinite execution is modeled via recursive process calls. In the `Router` process, first sub-process calls (e.g., `generateFlits`, `advanceRouter`) execute synchronously across all routers, then a recursive call to `Router` represents the next clock cycle. To model this non-terminating, recursive behavior in Dafny, we utilize an infinite while-loop (Code Segment 6.24), annotated with `decreases *` to suppress termination verification.

Code Segment 6.24. Infinite Loop Structure

```
method run(buffer_length: nat)
  decreases * // indicates method non-termination
{
  var routers := this.construct(buffer_length);

  while true
    decreases * { /* loop body */ }
}
```

Semantic Equivalence and Synchronization

Bridging the semantic gap between the concurrent process algebra of MODEST and the sequential execution model of Dafny is a critical component of this verification effort. Our task is simplified because the target MODEST model utilizes barriers to enforce strict lock-step execution for digital logic emulation. Consequently, we are not required to model arbitrary asynchronous interleavings; rather, we must guarantee that the global state transition produced by the sequential Dafny model is identical to the synchronized transitions in MODEST.

Boe [7, Sec. 5.1] demonstrated that fine-grained synchronization is not strictly required for NoC correctness. Specifically, placing synchronization barriers only at key phase boundaries–prior to packet injection (`generateFlits`) and routing (`advanceRouter`)–suffices to prevent race conditions (e.g., Write-Before-Read conflicts) and ensure correct packet propagation. In Dafny, we emulate these barriers by iterating sequentially over the entire set of routers for each distinct execution phase (as shown in Code Segment 6.10). Completing a loop over all routers for phase P before initiating phase $P + 1$ imposes a global barrier, effectively preserving the lock-step semantics of the hardware model.

However, modeling parallel composition via sequential iteration is only valid iff the operations within a given phase are *independent* (commutative). That is, *the state update of router R_i must not affect the input state of router R_j within the same phase.* Otherwise, the iteration order would dictate the result.

In our model, independence is strictly maintained. For internal phases (e.g., `generateFlits`, `updatePriority`), operations are strictly local to each router, making the execution order irrelevant. For the communication phase, `advanceRouter`, independence is preserved via the preceding `prepRouter` phase. `prepRouter` acts as a sampling stage, storing the necessary state (such as whether a buffer is full or empty) into local variables. `advanceRouter` then reads exclusively from these stored values to make routing decisions. By decoupling the "read" (sampling) logic from the "write" (routing) logic, we ensure that updates in the `advanceRouter` loop are commutative, rendering the sequential execution semantically

equivalent to the parallel MODEST model.

Assume-Guarantee Formulation

Although the MODEST model implicitly handles state transitions via process composition, Dafny requires explicit proofs of preconditions and postconditions. We adopt an AG reasoning style. Let Q represent the global NoC invariant and S represent the intermediate state between packet generation and packet routing. Code Segment 6.25 depicts a logical flow in the context of the MODEST process. In this flow, we do not require that S is related to Q, rather, the flow states that as long as Q holds before and after the execution of `Router`, the intermediate state does not have to imply Q. Additionally, in this flow, invariants are not shown after every line. This is only to simplify the Code Segment, but we could consider that after each process call we have a different invariant S_0, S_1, and so forth. The important take-away is that internal to a process, invariants need not be maintained, as long as they are maintained at the pre- and post-condition boundaries.

Code Segment 6.25. AG Reasoning in the MODEST Process

```
process Router(int id) {
  {∀r.r ∈ ℛ ⟹ Q}
  generateFlits(id);
  prepRouter(id);
  {∀r.r ∈ ℛ ⟹ S}
  advanceRouter(id);
  updatePriority(id);
  {∀r.r ∈ ℛ ⟹ Q}
  Router(id)
}
```

We implement this behavior in Dafny by sequencing loops, as shown in Code Segment 6.26. Each loop corresponds to a phase in the hardware cycle. By proving that the invariant Q holds before the first loop and is re-established after the final loop, we verify the correctness of the arbitrary $n \times n$ network.

Code Segment 6.26. Run Method with Sequential Phase Loops

```
method run(buffer_length: nat)
  decreases * // indicates method non-termination
{
  var routers := this.construct(buffer_length);

  while true
    decreases *
    invariant Q
  {
    {∀r.r ∈ R ⟹ Q}
    for i := 0 to |routers| { routers[i].generateFlits(); }
    for i := 0 to |routers| { routers[i].prepRouter(); }
    {∀r.r ∈ R ⟹ S}
    for i := 0 to |routers| { routers[i].advanceRouter(); }
    for i := 0 to |routers| { routers[i].updatePriority(); }
    {∀r.r ∈ R ⟹ Q}
  }
}
```

While this structure guarantees global synchronization, our verification strategy will further decompose these steps, capturing granular pre- and post-conditions for each sub-process call. This confirms that while Dafny does not execute in parallel, it formally verifies the same state-space transitions as the synchronous MODEST model.

Calculating $\mathcal{C}$ and Sending Packets

`advanceRouter` requires a valid set of neighbor router references as input. Formally, we derive these neighbors from the connection pairs $\mathcal{C}$ (Def. 2.4). In the MODEST model, we calculate these connection pairs statically before model checking and store them in the `ids` array of each router.

The first design we could adopt would be the addition of a constant set of sequences that hold the identifiers of the neighbors of each router–just like the MODEST model. However, as described in Section 6.6.6, this would introduce verification obligations to the object as a whole. Since Dafny excels at AG reasoning we will instead take an approach that keeps the modularity of the verification. *Our approach enables on-the-fly calculation of connections*

between routers each loop iteration, and keeps the neighbor verification obligations scoped to only the `advanceRouters` *call.*

Given that we have a square $n \times n$ mesh NoC mapped onto a sequence of routers and that the *id* of each router corresponds strictly to its index in the sequence, we use *row-major ordering* to determine router locations. This approach, common in systems programming languages like C, allows multidimensional sequences to be represented as a one-dimensional sequence. Consequently, using row-major ordering we can derive the location of a router and its neighbors solely from its *id*.

Definition 6.1. Coordinate Mapping. Given an $n \times n$ NoC and a router with a unique linear identifier $id \in [0, n^2 - 1]$, the Cartesian coordinates (x, y) of that router are calculated as:

$$x = id \pmod{n}, \quad y = \frac{id}{n}$$

where x represents the column index and y represents the row index.

Additionally, given an $n \times n$ NoC and a Cartesian coordinate (x, y) of a router in the NoC (i.e. $x, y \in [0, n)$) the identifier *id* of that router is calculated as:

$$id = x + (y \times n).$$

After establishing coordinate mapping, the initial implementation for determining neighbor identifiers explicitly converted between *id* and (x, y) coordinates. For example, to find a North neighbor identifier, the router would calculate the current (x, y), decrement y (move up a row), and re-apply the formula $x + (y \times n)$. The full calculation steps to find the north neighbor are

$$x = id\%n, \quad y = \frac{id}{n}, \quad y' := y + 1, \quad id_{north} := x + y' \times dim.$$

Expanding this out results in

$$id_{north} = id\%n + \left(\frac{id}{n} + 1\right) \times n.$$

This approach introduced arithmetic complexity—the division and modulo operations required to extract coordinates—which proved significantly challenging for the verification engine in Dafny. However, the row-major memory layout permits a much more efficient calculation using constant linear offsets. Because the routers are sequential:

- The *north* and *south* neighbors are located at offsets $id - n$ and $id + n$, respectively.
- The *west* and *east* neighbors are located at offsets $id - 1$ and $id + 1$.

This linear approach simplifies the proof burden significantly. We need only perform bounds checks (e.g., ensuring a router is not in the last column before requesting an East neighbor) and ensure the resulting index exists within the mesh. The optimized Dafny implementation is shown in Code Segment 6.27.

Code Segment 6.27. Neighbor Calculation using Linear Offsets

```
method getNeighborIds() returns (n: DirectionWrapper<Wrappers.Option<nat
    ↪ »)
  requires validMetadata()
  ensures fresh(n)
  ensures n.local.None?
  ensures
    var id := this.id - this.dim;
    && (isValidNeighborId(id) ⟹ n.north.Some? && n.north.value = id)
    && (!isValidNeighborId(id) ⟹ n.north.None?)
  ensures
    var id := this.id + 1;
    && (isValidNeighborId(id) && this.x() < this.dim - 1 ⟹ n.east.Some?
       ↪  && n.east.value = id)
    && (!isValidNeighborId(id) || this.x() = this.dim - 1 ⟹ n.east.
       ↪ None?)
  ensures
    var id := this.id + this.dim;
    && (isValidNeighborId(id) ⟹ n.south.Some? && n.south.value = id)
    && (!isValidNeighborId(id) ⟹ n.south.None?)
```

```
  ensures
    var id := this.id - 1;
    && (isValidNeighborId(id) && this.x() ≠ 0 ⟹ n.west.Some? && n.west
       ↪ .value = id)
    && (!isValidNeighborId(id) || this.x() = 0 ⟹ n.west.None?)
{
  n := new DirectionWrapper(Wrappers.Option.None, Wrappers.Option.None,
     ↪ Wrappers.Option.None, Wrappers.Option.None, Wrappers.Option.None
     ↪ );

  // North
  var id_n := this.id - this.dim;
  if isValidNeighborId(id_n) {
    n.north := Wrappers.Option.Some(id_n);
  }

  // East
  var id_e := this.id + 1;
  if isValidNeighborId(id_e) && this.x() < this.dim - 1 {
    n.east := Wrappers.Option.Some(id_e);
  }

  // South
  var id_s := this.id + this.dim;
  if isValidNeighborId(id_s) {
    n.south := Wrappers.Option.Some(id_s);
  }

  // West
  var id_w := this.id - 1;
  if isValidNeighborId(id_w) && this.x() ≠ 0 {
    n.west := Wrappers.Option.Some(id_w);
  }
}
```

For any given router in the mesh, after calculating the identifiers of its neighbors, we can use those identifiers to access the neighbor object in the `routers` sequence.

Introducing New Packets

As described in Section 6.6.4 the `generateFlits` method now expects the external environment (the caller) to produce valid packets and insert them into the local buffer of

the associated router. Code Segment 6.28 demonstrates how new flits are now injected into the network by `run`, while still using the same `getDestination` method from Sec. 6.6.4 to model non-determinism.

Code Segment 6.28. Generating New Packets in `run`

```
for i := 0 to |routers| {
  if cycle % 3 < 3 && routers[i].buffers.local.length() < routers[i].
      ↪ buffer_length {
    var dest := getDestination(routers[i].id, routers[i].dim);
    var dest_opt := Wrappers.Option.Some(dest);
    routers[i].generateFlits(dest_opt);
  }
}
```

Complete Run Function

The completed `run` method is shown in Code Segment 6.29.

Code Segment 6.29. Full Run Method

```
method run(buffer_length: nat)
  requires dim ⩾ 2
  requires buffer_length > 0
  decreases *
{
  var routers: seq<Router> := this.construct(buffer_length);

  var cycle: nat := 0;

  while true
    decreases *
    invariant 0 ⩽ cycle
    modifies routers[..]
    modifies (set x | x in routers :: x.serviced)
    modifies (set x | x in routers :: x.used)
    modifies (set x | x in routers :: x.buffers)
  {
    for i := 0 to |routers| {
      if cycle % 3 < 3 && routers[i].buffers.local.length() < routers[i].
          ↪ buffer_length {
        var dest := getDestination(routers[i].id, routers[i].dim);
        var dest_opt := Wrappers.Option.Some(dest);
```

```
      routers[i].generateFlits(dest_opt);
     }
    }

    for i := 0 to |routers| { routers[i].prepRouter(cycle); }

    for i := 0 to |routers| {
     routers[i].advanceRouter(routers);
    }

    for i := 0 to |routers| { routers[i].updatePriority(); }

    cycle := cycle + 1;
   }
  }
```

6.8 NoC Verification

To prove functional correctness of the NoC, we will prove that Properties 5.3, 5.4, 5.5, 5.6, and 5.7 hold on the Dafny model. This section covers the translation of these properties into Dafny, and also outlines a new property that verifies that all packets in the NoC are valid.

6.8.1 Translating Properties to Dafny

In the MODEST model, these functional correctness properties are encoded as CTL properties that operate directly on the generated state space of the model. Dafny has no concept of CTL, so we have to transform these into properties that are amenable to deductive verification.

Traffic Injection: Properties 5.3, 5.4 both cover traffic injection. However, Property 5.3 encodes a $\forall\Box$ (globally) invariant while Property 5.4 encodes a $\exists\Diamond$. Globally properties are much easier to encode, as they can be encoded as a simple invariant. Eventually properties are harder however. Leino encodes eventually using a combination of a schedule (which encodes the concept of time steps) and existential quantifiers in [65]. This approach is not amenable for this NoC model because of the choice of a flat class structure. Encoding a schedule relies on a transition function that can relate two states (s and s').

Since our states are encoded by the same object (i.e., s and s' reference the same router), we can't easily create a schedule. Instead we'll translate Property 5.4 to a more deductive friendly approach.

We'll formulate Property 5.3 as

$$d = getPacket(id, dest) \implies d \neq id \wedge d \in [0, dim^2). \tag{6.1}$$

This states that the postcondition of *getPacket* must return an id that is not equal to the input id.

We'll formulate Property 5.4 as

$$d = getPacket(id, dest) \implies d \in [0, dim^2) \tag{6.2}$$

which states the returned ID should be in the NoC. This is weaker than Property 5.4, which states that for all possible destinations in $[0, dim^2)$ there exists a path where one of those destinations is generated. By constrast, Property 6.2 only guarantees that the output is bounded by the possible destinations. This is weakened due to Dafny's lack of built-in CTL operators.

The next question to answer is *where* should we verify these properties in our model. Because MODEST is a model checker, these properties are checked on the state-space of the entire model. However, in Dafny, we can actually just check these properties on the traffic injection method because of the AG reasoning model. This boils down to checking whether the postcondition of the *getPackets* method (Code Segment 6.30) satisfies Properties 6.1 and 6.2.

Code Segment 6.30. Traffic Injection Verification

```
method testTrafficeInjection(id: nat, dim: nat)
  requires dim ⩾ 2 && 0 ⩽ id < dim*dim
{
  var destination := getPackets(id, dim);
  // No flits for self
  assert destination ≠ id;
  // All destinations possible
  assert 0 ⩽ destination < dim*dim;
}
```

Valid Priority List: Property 5.5 states the priority list is always valid. To prove this property, we can construct an invariant that states that each direction (four cardinal directions and local) are always present in the list. The proof that the length of the list is always five is handled by the `FixedSeq` subtype constraint and is automatically verified throughout the program. This invariant is shown in Code Segment 6.31.

Code Segment 6.31. Priority List Invariant

```
ghost predicate priorityListIsvalidMetadata()
  reads this`priority_list
{
  && North in this.priority_list
  && East in this.priority_list
  && South in this.priority_list
  && West in this.priority_list
  && Local in this.priority_list
}
```

The invariant in Code Segment 6.31 is stated slightly differently than the invariant in Code Segment 6.19. This is because `priorityListIsValid` ensures that each of the cardinal directions and local are in the priority list, and then because the length of the list is known to be five, it is implicit that each element in the list is unique. Dafny cannot automatically deduce that the invariant in Code Segment 6.19 is the same as `priorityListIsValid`, so we have to write a lemma to prove that.

The lemma (Code Segment 6.32) states that if the input sequence has a length of five

and that each element in the sequence is unique, then all five of the buffer directions are present in the sequence. While this lemma may be obvious to the reader, Dafny has no basis for reasoning about the user-defined `Direction` type, so we must provide proof that this lemma holds.

The lemma is structured by first capturing the elements of the sequence in a set, then asserting that the cardinality of this set is five. Dafny proves this from the precondition which states that every element in the sequence is unique.

Next, the lemma captures a set of all five buffer directions. This can be thought of as the "goal set", as we wish to prove that the set of our sequence is equal to this. This goal set provides context to the solver to help it prove the next point.

Finally, we complete the proof using a proof of contradiction. This proof method was chosen because it cleanly breaks down the problem into separate cases. The first case is proving that if $North$ is not present in our sequence, there is a contradiction. We structure this as a branch, where in the branch $North$ is not present in our sequence. If $North$ is not present, then the set of our sequence must be a subset of $\{East, South, West, Local\}$ (the valid directions without $North$). That would indicate that the cardinality of our set is at most four. Earlier, we proved that it was five. Therefore, we can `assert false`, which indicates to the Dafny verifier that this code path can never be reached, meaning that the original statement leading to this code path (`North !in items`) is a contradiction.

After proving this one contradiction, the Z3 SMT solver underlying Dafny was able to automatically prove the rest of the lemma.

Code Segment 6.32. Priority List Invariant Lemma

```
lemma priorityListAllDirectionsPresent(p: seq<Direction>)
  requires |p| == 5
  requires forall i, j | 0 <= i < 5 && 0 <= j < 5 && i != j :: p[i] != p
      ↪ [j]
  ensures North in p && East in p && South in p && West in p && Local in
      ↪  p
{
  var items := {p[0], p[1], p[2], p[3], p[4]};
  assert |items| == 5;

  var allDirs := {North, East, South, West, Local};
  assert |allDirs| == 5;

  if North !in items {
    assert items <= {East, South, West, Local};
    assert |items| <= 4;
    // Proof of contradiction
    assert false;
  }

  // Only prove North, then Z3 can automatically prove the rest
}
```

To verify this property for the model, we need to ensure that all routers in the NoC verify this property. We do this by adding our invariant `priorityListIsValid` to our loop invariant in run. This guarantees that for each step of our model, we have a valid priority list. Additionally, to prove this property globally, each method that modifies the priority list (`updatePriority`) needs to prove that after updating the priority list this invariant still holds.

Buffer Length: Property 5.6 encodes the property that the buffer length never exceeds the specified maximum, or in other words, that when a buffer is full the routing logic applies backpressure and doesn't transfer new flits into that buffer. In Dafny, this is encoded as an invariant for each class member as Code Segment 6.33. This invariant checks that all buffers are bounded by the specified buffer length.

Code Segment 6.33. Buffer Length Invariant

```
ghost predicate bufferLengthsvalidMetadata()
  reads this.buffers
{
  && this.buffers.north.length() ⩽ this.buffer_length
  && this.buffers.east.length() ⩽ this.buffer_length
  && this.buffers.south.length() ⩽ this.buffer_length
  && this.buffers.west.length() ⩽ this.buffer_length
  && this.buffers.local.length() ⩽ this.buffer_length
}
```

To verify this property across the whole NoC, we will add it as a loop invariant in the Run method and state that this property always holds for all routers in the NoC as shown in Code Segment 6.36. Additionally, we will need to prove that each function that modifies the buffers (`generateFlits`, `AdvanceRouter`) does not allow for buffer overflow.

Channel Usage: In the MODEST model, we verify channel usage by instrumenting the model with additional counters that count how many times each channel has been used. Essentially, these act as an additional proof that no channel was used twice, but they really are verifying that each channel is (a) marked as used if it is used, and (b) if it has been used, it is not used again. The instrumentation method works well for CTL model checking, because we can explicitly check all possibilities. However, for deductive AG reasoning, we would have to encode a precondition for this and a postcondition.

The precondition and postcondition would essentially become

$$channel.used \iff channel.count = 1 \land \neg channel.used \iff channel.count = 0.$$

Then our process definition would essentially include an extra line

```
if !used[channel] {
   used[channel] := true;
   count[channel] := count[channel] + 1;
}
```

Given the preconditions, this essentially becomes a tautology. Because of this, we forgoe verifying this property on the Dafny model, and rely soley on the implementation to provide this correctness.

For future work, another approach to this would be verifying that within one clock cycle no buffer grows by more than 1 element. However, this would require encoding a way to reason about the NoC state between *discrete time steps*, which does not currently exist.[2]

Valid Packets: An additional property that we wanted to prove was that all packets in the NoC at once were valid (Property 6.3). This property is technically proven by the properties that demonstrate correctness of traffic injection, but that does not cover the possible case that the router implementation accidentally changes a packets value. This property is encoded as

$$\forall p : p \in \mathcal{N} \implies 0 \leq p < dim^2 \tag{6.3}$$

where $p \in \mathcal{N}$ refers to all packets p present in the NoC $\mathcal{N}$ at any given time.

Initially, we encoded this as Code Segment 6.34 and used this predicate as a loop invariant in the run method and as a precondition and postcondition in all methods that updated the buffers in a router.

[2]We can track clock cycles in our model with a integer variable. However, the AG reasoning of Dafny turns this variable into a formula (e.g. $0 \leq cycles$) so we cannot use it to reason about the relationship between two states. Leino resolves this with a transition function that models the relationship between s and s' [65], however, we do not use the same modeling approach.

Code Segment 6.34. Initial Implemenation Checking All Packets Are Valid

```
ghost predicate allPacketsArevalidMetadata()
  requires validMetadata()
  reads this.buffers
{
  && (forall j | 0 <= j < this.buffers.north.length() :: isValidId(this.
     ↪ buffers.north.buffer[j]))
  && (forall j | 0 <= j < this.buffers.east.length() :: isValidId(this.
     ↪ buffers.east.buffer[j]) )
  && (forall j | 0 <= j < this.buffers.south.length() :: isValidId(this.
     ↪ buffers.south.buffer[j]))
  && (forall j | 0 <= j < this.buffers.west.length() :: isValidId(this.
     ↪ buffers.west.buffer[j]) )
  && (forall j | 0 <= j < this.buffers.local.length() :: isValidId(this.
     ↪ buffers.local.buffer[j]))
}
```

This approach, however, proved to be fatal for our verification time. Possibly it has to do with the highly nested universal quantifiers, i.e., $\forall r : \forall b : \forall p : \ r \in \mathcal{N} \wedge b \in r \wedge p \in b \wedge 0 \leq p < dim^2$. Regardless if it was the nested quantification or not, this change took verification times from less than 30 seconds to timeouts occuring at 5 minutes. That does not mean that at 5 minutes it failed to verify, rather, the solver just stopped trying. When this occurs in Dafny, it likely means that some part of verification approach is incorrect or inefficient.

To resolve these timeouts, our second approach used a common Dafny technique: introduce a ghost (not compiled) variable that represents a set of states in the model. We introduced the `all_packets` set, which represents the set of all packets in the NoC at any time. Now our property can be encoded as Code Segment 6.35. This property is much easier for Dafny to check. To ensure this property is correct, we ensured that our representation of all packets in the NoC is valid. This is ensured by adding any packets introduced into the router (either in `generateFlits` or `advanceRouter`) to the `all_packets` set. Then in every method that modifies any buffer, we add `allPacketsAreValid` as a precondition and postcondition. Finally, we add `allPacketsAreValid` as a loop invariant in our verified run loop (Code Segment 6.36) to ensure it always holds.

Code Segment 6.35. Initial Implementation Checking All Packets Are Valid

```
ghost predicate allPacketsArevalidMetadata()
  reads this`all_packets
{
  forall p | p in this.all_packets :: isValidId(p)
}
```

After introducing each of these properties into the run loop we can fully verify the properties covered in Section 5.2, as shown in Code Segment 6.36.

Code Segment 6.36. Run Loop With Verification

```
while true
  decreases *
  invariant 0 ≤ cycle
  invariant forall r | r in routers :: fresh(r) && fresh(r.buffers) &&
      ↪ fresh(r.serviced) && fresh(r.used)
  invariant forall r | r in routers :: r.bufferLengthsvalidMetadata() &&
      ↪  r.priorityListIsvalidMetadata() && r.allPacketsArevalidMetadata
      ↪ ()
  modifies routers[..]
  modifies set x | x in routers :: x.serviced
  modifies set x | x in routers :: x.used
  modifies set x | x in routers :: x.buffers
  modifies set x | x in routers :: x`all_packets
{
  for i := 0 to |routers|
    invariant forall r | r in routers :: r.bufferLengthsvalidMetadata()
        ↪ && r.priorityListIsvalidMetadata() && r.
        ↪ allPacketsArevalidMetadata()
  {
    if cycle % 3 < 3 && routers[i].buffers.local.length() < routers[i].
        ↪ buffer_length {
      var dest := getPackets(routers[i].id, routers[i].dim);
      var dest_opt := Wrappers.Option.Some(dest);
      routers[i].packetsInBufferAreValidAxiom(Local);
      routers[i].generateFlits(dest_opt);
    }
  }
  for i := 0 to |routers|
```

```
    invariant forall r | r in routers :: r.bufferLengthsvalidMetadata()
        ↪ && r.priorityListIsvalidMetadata() && r.
        ↪ allPacketsArevalidMetadata()
  {
    routers[i].prepRouter(cycle);
  }

  for i := 0 to |routers|
    invariant {:split_here} forall r | r in routers :: r.
        ↪ bufferLengthsvalidMetadata() && r.priorityListIsvalidMetadata
        ↪ () && r.allPacketsArevalidMetadata()
  {
    routers[i].advanceRouter(routers);
  }

  for i := 0 to |routers|
    invariant forall r | r in routers :: r.bufferLengthsvalidMetadata()
        ↪ && r.priorityListIsvalidMetadata() && r.
        ↪ allPacketsArevalidMetadata()
  {
    routers[i].updatePriority();
  }

  cycle := cycle + 1;
}
```

RESULTS & ANALYSIS

The following chapters outline the results of this thesis. Chapter 7 covers the quantitative PSN results and analysis from the MODEST NoC model. Chapter 8 covers the functional verification results and analysis of the MODEST NoC model (using CTL) and the Dafny NoC model. Chapter 9 covers potential future work.

CHAPTER 7

Quantitative Analysis of Power Supply Noise using MODEST

This chapter presents the quantitative PSN results obtained from SMC and PMC of the modular NoC model described in Chapter 4 and 5 and the results of checking the CTL properties from Chapter 5. The PSN analysis focuses on three areas:

- the impact of the synchronous execution model on state-space size,
- PSN characterization using the improved per-router metrics, and
- the influence of different traffic injection patterns on PSN.

All results were generated using the software environment detailed in Section 4.9 and are available in [69]. Additionally, all results were run using Ubuntu Linux version 22.04.05 LTS on a 12-core AMD Ryzen Threadripper Processor (at 3.5 GHz) with 132 GB of memory (126 GB available). MODEST TOOLSET v3.1.290 was used for all results.

7.1 Impact of Synchronization on State-Space

A primary contribution of the updated modular model was the introduction of synchronizing actions (Section 4.5.2) to model synchronous digital semantics and prevent interleavings of independent state updates. Previous iterations of the model [7] suffered from significant state-space explosion due to these interleavings.

Tables 7.1 and 7.3 compare the required *verification resources* (i.e., memory usage and time) between the model in [7] and the synchronized model for a 2×2 NoC. Table 7.1 shows the results for CTL model checking the correctness properties from Section 5.2. The CTL model checking was completed using MODEST's `mcsta` model checking engine with the `-unsafe` and `-chainopt` flags set. The `-unsafe` flag disables some runtime checks, enabling faster performance. The `-chainopt` flag collapses transistions of probability 1.0, automatically reducing the state space.

The introduction of lock-step synchronization resulted enabled successful CTL model

Table 7.1: Comparison of Verification Resources for Functional Correctness

(CTL Checking 2×2 NoC with `mcsta -unsafe -chainopt`)

Model Version	**States**	**Memory (GB)**	**Time (s)**
Boe [7]	Memout	≥ 126	104621
Proposed Synchronized Model	3446073	9.24	546
Reduction from synchronization:	–%	$\geq 92.6\%$	$\geq 99.4\%$

checking for the 2×2 NoC with our current verification resources. The verification attempt of the previous 2×2 NoC model took approximately 29 hours, and ran out of memory before completing. The synchronized model, however, completed verification in about 10 minutes (≈190x speed-up) and only used 9.24 GB of RAM. This demonstrates how powerful the synchronized model is for model checking, and suggests that model checking could be further scaled using the synchronized model.

This massive improvement also highlights the benefits of modeling only the semantics seen in the system. The asynchronous interleavings present in the model did not represent the real NoC behavior, and also rendered model checking intractable, forcing the usage of SMC. Adding the additional coding complexity of the synchronizing actions resolved this issue, and made explicit state enumeration model checking tractable for small NoC models.

Table 7.2 shows the comparison of verification resources between the abstract NoC model and the asynchronous model from [7]. This isn't a fair comparison, as the architecture of the two models is completely different, with the asynchronous model containing four full router instances, and the abstract model containing one full router instance and four abstract routers. However, the results show that the abstract verification is again at the limit of our computational resources and that without additional abstraction cannot be scaled efficiently.

Table 7.3 shows the required verification resources needed for inductive PSN analysis using SMC using Property 2.7 where $N \in [0, 100]$ and $K = 10$ with the following NoC setup:

- buffer size of 4,
- activity threshold of 3, and

Table 7.2: Verification Resources for Functional Correctness on the Abstract Model

(CTL Checking abstract NoC with `mcsta -unsafe -chainopt`)

Model Version	**States**	**Memory (GB)**	**Time (s)**
Boe [7]	Memout	≥ 126	104621
Proposed Abstract Model	77001660	135	152284
Reduction from synchronization:	–%	$\geq -8\%$	$\geq -45.6\%$

- the X-Y routing, Round-Robin arbitration, and 3/10 traffic injection model discussed in Chapters 4.

SMC was run using MODEST's `modes` statistical model checking engine with the `-unsafe` flag set and `-max-run-length` set to zero. `-unsafe` disables some runtime bounds checking, enabling faster analysis, and `-max-run-length` specifies that there is no maximum number of steps per run. Additionally, in the results shown in Table 7.3, the `-D` flag was used to extract additional resource utilization information for comparison. This flag was not used for other PSN results.

Table 7.3: Comparison of Verification Resources for PSN Characterization

(PSN of 2×2 NoC with global PSN characterization up to 100 cycles with $\sigma = 0.95$)

Model Version	**# of Traces**	**Mean Trace Length**	**Memory Usage**	**Time (s)**
Boe [7]	9794	6237.8	121 MB	59.4
Proposed Model	9841	2104.1	107 MB	28.3
Reduction:	-0.5%	66.2%	11.5%	52.3%

The introduction of synchronizing actions also provided a performance increase in SMC runs. While the number of traces generated by the `modes` tool is roughly the same, the average length of each trace greatly differs, with the proposed synchronizing model having, on average, traces that are 66% shorter. Additionally, the shorter traces saved memory and reduced runtime. These performance benefits help improve the scalability of PSN characterization with the modular NoC model.

7.2 Power Supply Noise Results

This section details extended PSN characterization that builds off the work of [7].

7.2.1 Per-Router PSN Characterization

As opposed to previous approaches that tracked global noise events, this work implemented per-router activity tracking (Section 5.1.2). This granularity allows for the identification of "hotspots" within the mesh topology.

Resistive Noise Distribution

Figure 7.1 shows a heatmap distribution of resistive activity across an 8×8 NoC through clock cycle 10. Visually, this distribution is expected as it shows that routers with more neighbors (i.e., non-edge routers) have a higher probability of seeing three or more events in one clock cycle by cycle 10. It also shows (on the bottom row) that horizontal traffic is more likely to occur than vertical traffic. Due to the X-Y routing implementation, this is also to be expected.

Figure 7.2 shows the heatmap distribution for the resistive activity of the 8×8 router over clock cycles 1, 5, and 10. Examining the sequence of these heatmaps confirms our expectations of this router design, namely that due to X-Y routing and uniform packet injection, we see a uniform increase in resistive PSN events focused on the central routers. This suggests that corner routers are far less likely to experience PSN events, and can likely be optimized to use less power than central routers.

Inductive Noise Analysis

Figure 7.3 shows a heatmap distribution of inductive activity (the change in activity across cycles) for an 8×8 NoC through clock cycle 10. Immediately, the visual appearance of the noise is quite different to that of the resistive noise in Figure 7.1. While the probability of resistive noise activity was centered in the NoC, the noise of this router appears to instead be concentrated in the center of each 4×4 quadrant of the NoC. Routers 9, 14, 49, and 54 have the highest probability of inductive activity across the NoC. Re-examining the resistive

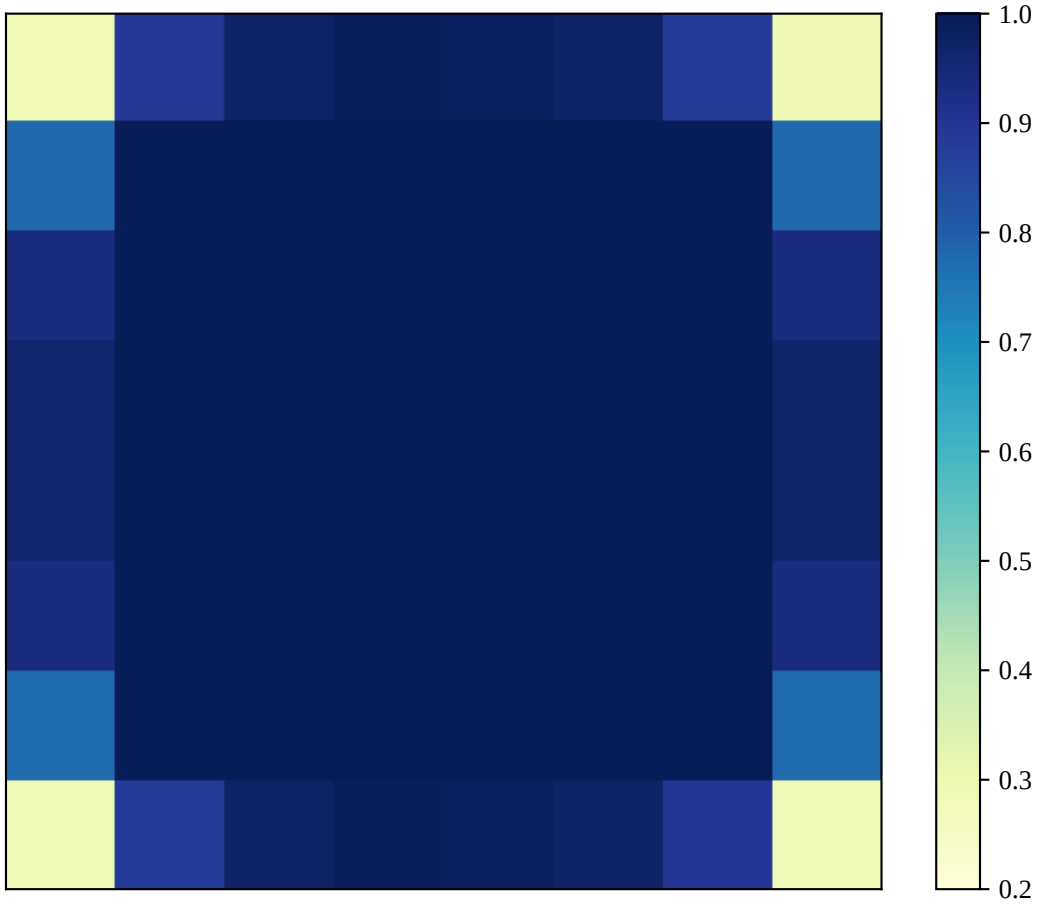
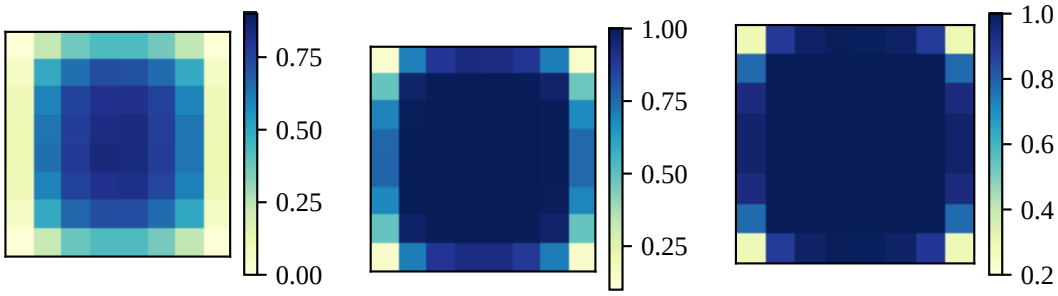


Fig. 7.1: Probability of Resistive Activity per Router Exceeding Two at Cycle 10. (8×8, Uniform 3/10 Traffic, X-Y Routing).

noise activity, this result is logical because those are the areas where the largest difference between the resistive activity of two neighbors. The central routers have high activity, but they are consistently active, and the edge routers have consistent low activity, so it's the routers in-between these two regions that experience the worst inductive noise activity.

Figure 7.4 shows the inductive activity heatmap over clock cycles 5, 10, and 50. Examining this sweep confirms our previous conclusion. We can see that the probability of inductive noise events occurring at the corner routers between the central routers and the edge routers is consistently higher than other routers in the NoC. Another observation is that corner routers are once again the least likely to experience inductive noise events, again likely leading to optimization opportunities in physical design.



(a) Clock Cycle 1 (b) Clock Cycle 5 (c) Clock Cycle 10

Fig. 7.2: Resistive Activity Heatmap Three Across Different Clock Cycles (8×8, Uniform 3/10 Traffic, X-Y).

Figure 7.5 shows the inductive activity through clock cycle 10 on a 4×4, 8×8, and 12×12 NoC with 3/10 uniform traffic and X-Y routing. This comparison of differently sized NoCs reveals a trend highlighting local inductive activity hotspots around each of the four quadrants of the NoC. This trend is not visible in smaller NoC analysis (e.g., Figure 7.5a), and 11/16 of the real-world NoCs used in [9] are between sizes 4×4 and 14×14. This highlights the usefuleness of the scalability of this method for analyzing real-world NoC designs.

7.2.2 Traffic Pattern Analysis

To evaluate the robustness of the NoC design, we compared the uniform 3/10 traffic injection model to the bursty model presented in Section 4.19.

- **Uniform:** Packets injected with a 0.3 duty cycle every cycle.
- **Bursty:** Periods of high injection ($10-12$ packets) followed by silence.

Figure 7.6 shows the resistive noise activity CDFs per router in a 2×2 NoC for both traffic injection patterns. These CDFs demonstrate that the bursty injection pattern nearly immediately causes the probability of a resistive noise event to occur in every router in the NoC. This matches our expectations, as the bursty pattern quickly attempts to inject $40-48$ packets into the network as fast as possible. This occurs because each router injects $10-12$ packets after a short sleep period at the beginning. As a reminder, the probability stays at 1.0 as this is cumulative distribution. These results suggest that the uniform traffic

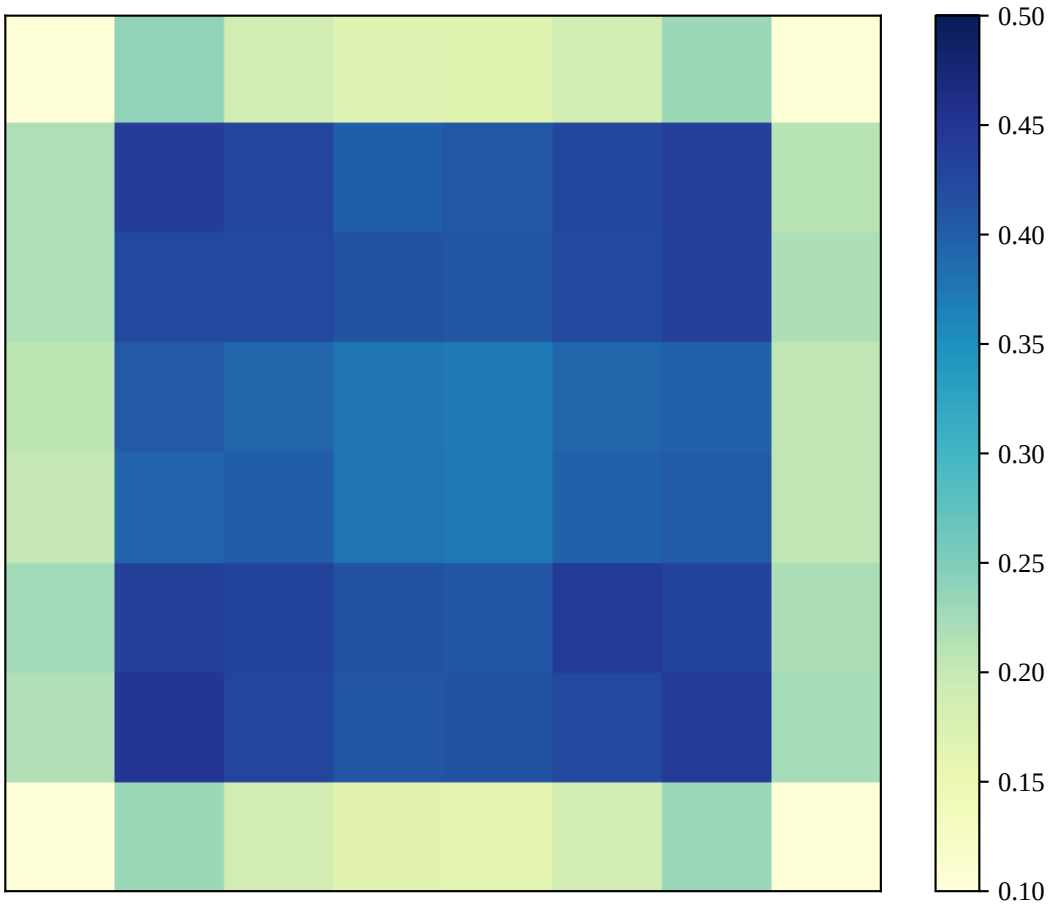
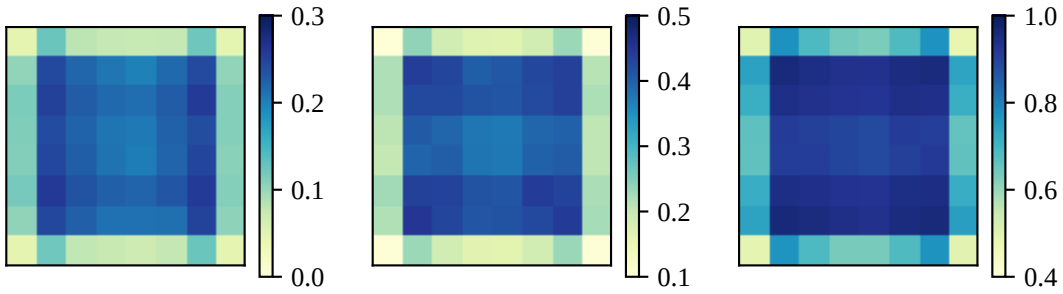


Fig. 7.3: Probability of Inductive Activity per Router Exceeding Two at Cycle 10. (8×8, Uniform 3/10 Traffic, X-Y Routing).

injection of $[7, 8]$ does not represent a worst-case real-world scenario for resistive activity.

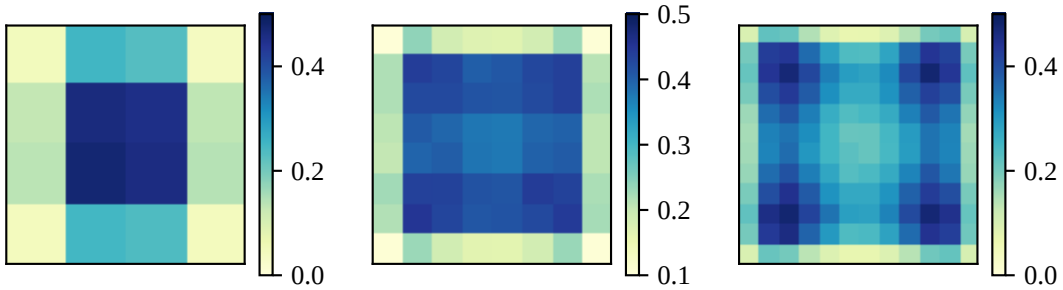
Interestingly, the inductive noise comparison shown in Figure 7.7 showcases an opposite scenario. The uniform traffic pattern significantly generates more inductive noise events in each router with the maximum probability approaching 0.75 at 1000 clock cycles. Contrastingly, the bursty pattern displays a gentle step increase during each bursty phase, but after 100 clock cycles the maximum probability is only approaching 0.2. This indicates that the bursty traffic pattern may not be as susceptible to inductive PSN as the uniform traffic pattern.

However, this difference in inductive activity between the 3/10 and bursty traffic pattern may be because the implementation of bursty pattern roughly sees each router execut-



(a) Clock Cycle 5 (b) Clock Cycle 10 (c) Clock Cycle 50

Fig. 7.4: Inductive Activity Heatmap Across Three Different Clock Cycles (8×8, Uniform 3/10 Traffic, X-Y).



(a) 4×4 (b) 8×8 (c) 12×12

Fig. 7.5: Inductive Activity Heatmap Through Cycle 10 On Different Sized NoCs (Uniform 3/10 Traffic, X-Y).

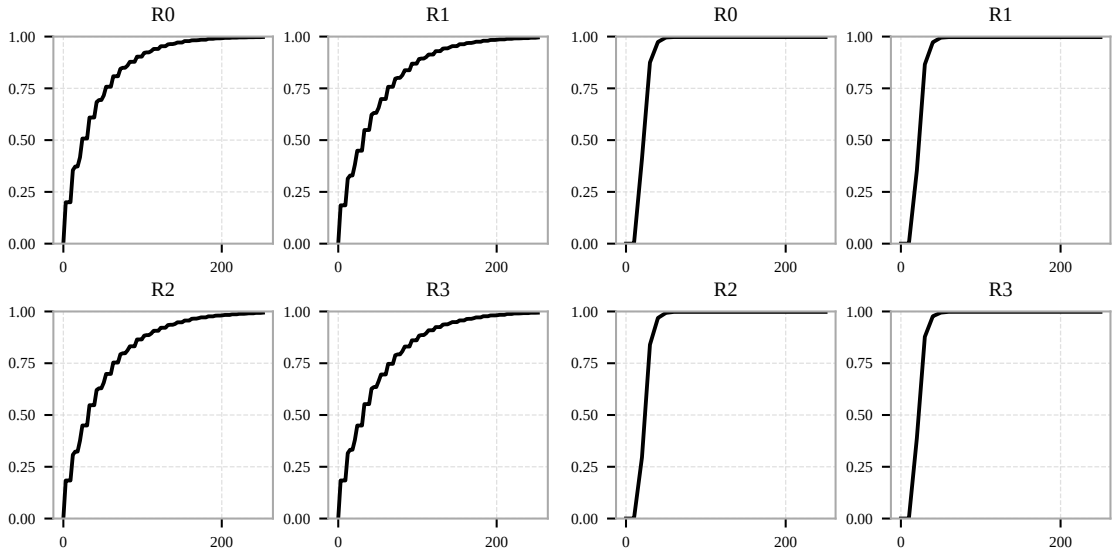
ing a burst at the same time. Future work could analyze alternate bursty implementations as described in Chapter 9.

The results in this section highlight the importance of traffic pattern modeling. Recent work described in [71] demonstrates that traffic patterns in NoCs are not well understood and presents several possible traffic patterns for analysis.

7.3 Impact of NoC Size on Verification Time

Table 7.4 shows the corresponding time it took to compute PSN characterization for different sized NoCs using the synchronized model. Results are shown for the time taken to compute the PSN characterization of four different setups:

1. resistive noise activity characterization per router (Per Router, Res),



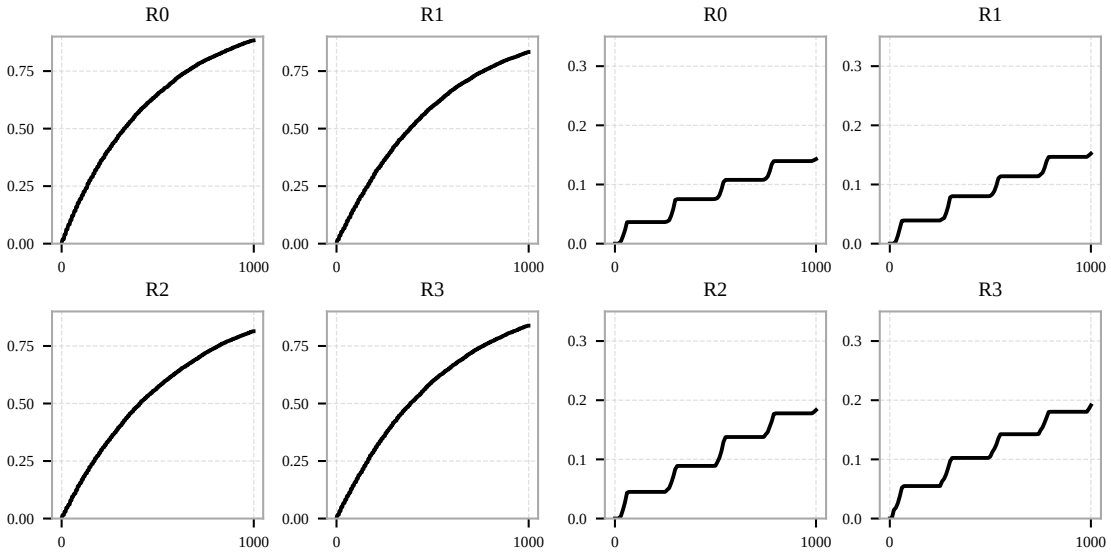
(a) Uniform 3/10 (b) Bursty

Fig. 7.6: Resistive Activity Comparison Between Uniform and Bursty Traffic.

2. inductive noise activity characterization per router (Per Router, Ind),
3. global resistive noise characterization (Global, Res), and
4. global inductive noise characterization (Global, Ind)

where the "global" characterization is the PSN method presented in [7, 8] and "per-router" is the method presented in Section 5.1.2.

These results cannot be directly compared cleanly, as the global verification contains additional parameters not present in the per-router verification. However, the results suggest that for inductive noise, the global measure is often about than 50% faster than the per-router model, while for resistive noise the opposite appears between the global and the per-router measurements. However, the verification time between the inductive and resistive global measurements appear to be relatively similar, suggesting that global verification time is similar whether or not inductive or resistive noise is considered. For per-router measurements, inductive noise is much more time consuming than resistive noise characterization.



(a) Uniform 3/10 (b) Bursty

Fig. 7.7: Inductive Activity Comparison Between Uniform and Bursty Traffic.

Table 7.4: Verification Time Using SMC Per NoC Size

NoC Size	**Time (s)**			
	Per Router, Ind	Global, Ind	Per Router, Res	Global, Res
2×2	237.9	–	31.1	35.5
4×4	1110.5	671.5	238.0	626.5
8×8	11639.6	4633.6	2605.5	4418.2
12×12	33421.5	17414.7	8570.8	16957.4

7.4 Review of Results

This section presented the results of the MODEST verification results of this thesis. Our work demonstrated a significant improvement in verification resources through our introduction of synchronizing actions. This improvement led to the ability to complete CTL model checking of the properties outlined in Section 5.2 on a 2×2 NoC. We additionally verified these CTL properties on an abstract NoC that extensively stimulated a single router instance. We extended the PSN characterization results of [7] to larger size NoCs, as well as more diverse traffic patterns. In total, this thesis provided significant analysis and refinement of the NoC model presented in [7], and provided a path for future verification opportunities using this methodology.

CHAPTER 8

Functional Verification of the NoC Model

This chapter details the results of the functional verification of the MODEST NoC model using CTL and the Dafny NoC model using the Dafny verification engine. All properties from Section 5.2 were verified on a 2×2 MODEST NoC model and an $n \times n$ Dafny NoC model.

8.1 CTL Model Checking the 2×2 MODEST Model

All properties given in Section 5.2 were successfully verified using the MODEST CTL model checking engine, after discovering one small discrepancy in the results and resolving the modeling issue. When synchronizing each process (Section 4.5.2), some logical branches in the model were moved from outside a assignment block to be inside the assignment block, in order to facilitate cleaner synchronization. Shifting these logical branches required a close reading of the original model and careful translation from an `if(){} else{}` statement to an `if () then else` expression. The initial implementation of the `AdvanceChannel` process shifted a large block of code from an if-statement to an if-expression. A minor detail in the translation of this process caused the failure of Property 5.7 (each channel may only send packet per clock cycle) for ever local channel in every router.

Specifically, the second branch of the if-statement specifying `AdvanceChannel` describes the process of receiving a packet destined for the current router, or in other words, the behavior of a packet that has reached its destination. This branch is triggered when the front packet of a buffer has a destination identifier equivalent to the identifier of the current router. If this occurs, then it is checked if the local channel has already been used during this clock cycle. If the channel has been used, then the total number of unserviced routers is incremented, and nothing else happens. If it has not been used, then the buffer is marked as serviced, the packet is popped off the buffer and sent across the local channel, and the channel count is incremented once. This describes the process of receiving a packet.

Code Segment 8.1 shows the implementation of this from [7]. The first attempt to synchronize this (Code Segment 8.2) flattened the logic from the if-statement into several if-expressions using the ternary operator (`? :`) and setting the condition for each assignment to be whether or not the local channel had been used this cycle. This allowed this entire branch to be synchronized on one action, instead of multiple, further reducing the state-space and verification time.

However, two subtle bugs occurred. First, the counter tracking whether or not a packet was sent was accidentally left unguarded, resulting in the count incrementing whether or not a packet was sent across the channel. Secondly, because the local channel was not marked as "used" but rather as not previously used, this allowed for circumstances where multiple packets actually were sent across the local channel in one clock. This is not allowed in our model, and was caught by MODEST in the form of returning `False` for Property 5.7.

Code Segment 8.1. Original Implementation of Branch Two of Advance Channel [7]

```
// If front packet has the same id as this router,
// then we have received a packet destined for us
else if(peekFront(noc[id].channels[ch].buffer) == id){
   if (!noc[id].used[LOCAL]) {
      {=
      noc[id].channels[ch].serviced = true,
      noc[id].used[LOCAL] = true,
      noc[id].channels[ch].buffer = dequeue(noc[id].channels[ch].buffer)
          ↪ ,
      sendCounts[id].counts[LOCAL] = sendCounts[id].counts[LOCAL] + 1
      =}
   }
   else {
      {=
      noc[id].total_unserviced++
      =}
   }
}
```

Code Segment 8.2. Incorrect Synchronization of Branch Two of Advance Channel

```
else if (peekFront(noc[id].buffers[ch].buffer) == id) {
   advanceChannel {=
      1: noc[id].buffers[ch].serviced = !noc[id].used[LOCAL],
      1: noc[id].buffers[ch].buffer = !noc[id].used[LOCAL] ? dequeue(noc
         ↪ [id].buffers[ch].buffer) : noc[id].buffers[ch].buffer,
      1: sendCounts[id].counts[LOCAL] = sendCounts[id] + 1,
      1: noc[id].total_unserviced = !noc[id].used[LOCAL] ? noc[id].
         ↪ total_unserviced : noc[id].total_unserviced + 1,
      2: noc[id].used[LOCAL] = !noc[id].used[LOCAL]
   =};
}
```

After diagnosing the problem, branch two of `AdvanceChannel` was more explicitly rewritten to match the original behavior, as shown in Code Segment 8.3. This rewriting passed all properties when run through the CTL model checking engine.

Code Segment 8.3. Fixed Synchronization of Branch Two of Advance Channel

```
else if (peekFront(noc[id].buffers[ch].buffer) == id) {
   advanceChannel {=
      1: noc[id].buffers[ch].serviced =
         if !noc[id].used[LOCAL]
         then true
         else noc[id].buffers[ch].serviced,
      1: noc[id].buffers[ch].buffer =
         if !noc[id].used[LOCAL]
         then dequeue(noc[id].buffers[ch].buffer)
         else noc[id].buffers[ch].buffer,
      1: sendCounts[id].counts[LOCAL] =
         if !noc[id].used[LOCAL]
         then sat_add(sendCounts[id].counts[LOCAL], 1, 2)
         else sendCounts[id].counts[LOCAL],
      1: noc[id].total_unserviced =
         if !noc[id].used[LOCAL]
         then noc[id].total_unserviced
         else noc[id].total_unserviced + 1,
      2: noc[id].used[LOCAL] =
         if !noc[id].used[LOCAL]
         then true
         else noc[id].used[LOCAL]
   =};
}
```

This experience highlights the benefit of having a CTL model checking engine integrated with a PMC toolset. This issue, had it not been caught, would have incorrectly impacted the PSN analysis of the NoC model, as multiple packets could have been sent to the PE each cycle. This would have both increased resistive activity on cycles where multiple packets were available to be consumed, and also would have had an impact on overall PSN as packets could be removed from the network more quickly than expected. Because of CTL model checking, we were able to catch this issue early and provide correct PSN analysis.

8.2 CTL Model Checking the Abstract Model

All properties given in Section 5.2 were successfully verified using the MODEST CTL model checking engine on the abstract model. The successful verification of these properties

provides a high confidence the router process is functionally correct and safe. These results, however, do not ensure that all NoC instantiations are correct, as the connections between routers and traffic patterns need to be verified. However, since each router in the NoC is functionally equivalent, the abstract verification provides high confidence that the core safety properties will be maintained in the instantiated NoC.

8.3 Verification of NoC Construction

The initial state the Dafny model is generated by the `construct` method (Section 6.7.1). We successfully proved that for any dimension $dim \geq 2$, the method produces a valid set of routers $\mathcal{R}$.

1. **Property:** The `construct` method ensures $|routers| == dim \times dim$ and all routers are strictly unique, valid, and in an initial state (i.e., empty buffers, cleared flags, and devoid of packets).
   - ✓ **Verified**. This proof guarantees that the initial state of the system is always correct by definition, preventing "initialization bugs" from propagating into the logic verification.

8.3.1 Safety Properties

We formally verified the following safety invariants across the execution of the `run` method:

1. **Traffic Injection Validity** (Properties 6.1 and 6.2): this property states that packets are never generated for the router they originate from, and that all possible destinations may be generated.
   - ✓ **Verified.** This property was verified directly on the `getPackets` method, which was used as the new packet generator in the model.
2. **Priority List Validity** (Property 5.5): this property states the priority list is always in a valid state.
   - ✓ **Verified.** This property was verified as a per-router invariant in the `Router` class and `run` method.

3. **Bounded Buffer Length** (Property 5.6): this property states that the specified buffer length is never exceeded.

   ✓ **Verified.** This property is verified as a per-router invariant for both packet injection and transmission. Additionally, it is proven for any buffer size $b \geq 1$.

4. **Channel Usage** (Property 5.7): this property states that each channel is only to used to transmit a maximum of one packet per clock cycle.

   ∼ **Verification not attempted.** As described in Section 6.8, to verify this property using deductive verification would have created a tautological verification loop, as the verification procedure would have matched the implementation. This highlights the difference between deductive verification and model checking, as this property caught a bug in the MODEST model, which only occured because the proof and implementation were separate (Section 8.1).

5. **Packet Validity** (Property 6.3): this property ensures that every packet at all times in the NoC is correct.

   ✓ **Verified.** This property is verified on a phantom verification member of each router instance, `all_packets`, that contains all packets in the NoC at any time.

8.4 Verification Effort and Metrics

To quantify the effort required for deductive verification, we analyze the code metrics of the Dafny implementation. Table 8.1 breaks down the lines of code required to specify the generic NoC against the MODEST NoC model. This analysis was done using the `cloc` command line tool to cound lines of code ignoring blank lines and comments.

The analysis in Table 8.1 suggests that, in general, more lines of code are needed to implement the deductive NoC. However, this does not capture the nuance that in the Dafny model, proofs are contained in-line with the implementation, while in the MODEST model they are separated out. Additionally, Table 8.1 does not capture some of the additional tradeoffs of each model, e.g., the 116 CTL properties specified for the MODEST model are only the properties for a 2×2 NoC. Since MODEST does not have quantification over

properties, there is not a concise way to write a property such as

$$\forall R_i.B_{local} \in \mathcal{N} : |R_i.B_{local}| \leq BUFFER\_LENGTH,$$

leading instead to that property being duplicated on a new line for each i.

However, despite Table 8.1, only presenting a weak indication of the relation between the two models, it is clear that the Dafny model requires more lines of code, and more explicit annotations describing the correct operation of the NoC. Many operations in the MODEST model are assumed (in the model) to be correct, and only during state-space exploration is it determined if they actually are correct. In the Dafny model, however, there can be no assumptions about correctness, it must be explicitly proven that each method is correct.

Table 8.1: Lines of Code (LOC) for the MODEST and Dafny Models.

Component	**Modest LOC**	**Dafny LOC**	**Ratio** [MODEST:Dafny]
Data Structures	38	140	[1:3.7]
Router Logic	231	417	[1:1.8]
NoC Construction	21	111	[1:5.3]
CTL properties	116	–	[–]
Total	**406**	**668**	**[1:1.6]**

Once the model is complete, however, the verification using the AG framework in Dafny via preconditions and postconditions verifies remarkably fast. The verification time for the complete model on a 2023 Apple MacBook Pro with a Apple M3 Pro Chip and 18 GB of RAM is approximately 7 seconds. Notably, this verification time is independent of the NoC dimension n, as the proof holds for the generic parameter dim. This stands in stark contrast to the verification time of the MODEST model.

8.5 Comparison: Model Checking vs. Deductive Verification

The complementary nature of this thesis's two approaches is highlighted in their scalability results. While the MODEST model provides deep quantitative insights into PSN, for

functional correctness, it is limited to small topologies (2×2). The Dafny model, conversely, provides no probabilistic data but guarantees functional safety for any $n \times n$ mesh. This confirms that important NoC verification can occur using either MODEST or Dafny, but neither tool provides a all-in-one solution for NoC verification.

CHAPTER 9

Future Work

This chapter outlines potential future work the MODEST and Dafny NoC models.

9.1 General Improvements

Both the MODEST and the Dafny model use looping sequential logic to model digital hardware. Future work should compare these implementations to a hardware NoC design with matching semantics implemented using a *hardware description language* (HDL), which is the tool used to describe real computer chips. A comparative approach between a HDL implementation and the MODEST and Dafny implementation could reveal areas where the models are incorrect and provide additional confidence in the functional correctness and PSN characterization done by the MODEST and Dafny models.

9.2 Modest model

After several years of work the MODEST model is performant, correct, and easy to use. However, there are several additional aspects that could be explored in future work:

1. Implementation of unique traffic patterns a la [71].
2. Verification where the NoC starts in a specific state besides the empty state.
3. Verification of validity of single flit packet abstraction.
4. Verification of activity abstraction, and correlation between activity abstraction and real values.

9.3 Dafny model

The Dafny model is new, but showcases the usefulness of deductive verification. Future work could look at compiling it into a simulator executable and instrumenting that simulator

to run traces. Additionally, liveness proofs such as “a flit eventually arives to its destination” or “every router is fair” could be attempted.

CHAPTER 10

Conclusion

This thesis presented three novel contributions to the formal verification of NoC technology. These contributions improve NoC reliability and verification techniques. The contributions consist of (1) improvements to the PSN characterization of the MODEST NoC model presented in [7], (2) proof of functional correctness of the MODEST NoC model using CTL, and (3) introduction and verification of a NoC model in the Dafny language.

The MODEST NoC model was improved by introducing synchronizing actions to limit the statespace and more correctly model digital synchronous hardware. This change enabled verification of the functional correctness of the model using CTL. Regarding PSN, model changes were presented to allow per-router PSN characterization across the NoC. Additionally, an abstracted NoC router was constructed to give high confidence of the functional correctness of the model for arbitrary $n \times n$ NoCs. These changes together provide high confidence in the functional correctness of the NoC and demonstrate how the MODEST model can be used to optimize PSN.

In addition to extending the MODEST model, we created a NoC model semantically similar to that of the MODEST model using the Dafny language. This model mimics the parallel NoC behavior in a sequential language through careful construction of the NoC data structure and operation. The same functional correctness properties that were verified on the MODEST model were verified against the Dafny model. Unlike the MODEST model, these properties were verified for arbitrary $n \times n$-sized NoCs. This demonstrated the capability of Dafny to quickly verify large scale systems using deductive verification. Also, this model acts as a blueprint for future hardware verification in Dafny.

Together, these contributions can be used to improve NoC design and implementation on computer chips and will lead to future research opportunities in this field.

Bibliography

[1] J.-J. Lecler and G. Baillieu, "Application driven network-on-chip architecture exploration & refinement for a complex SoC," *Design Automation for Embedded Systems*, vol. 15, no. 2, pp. 133–158, Jun. 2011.

[2] W.-C. Tsai, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen, "Networks on Chips: Structure and Design Methodologies," *Journal of Electrical and Computer Engineering*, vol. 2012, no. 1, p. 509465, 2012.

[3] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz, "A Formal Approach to the Verification of Networks on Chip," *EURASIP Journal on Embedded Systems*, vol. 2009, no. 1, p. 548324, 2009.

[4] P. Basu, R. Jayashankara Shridevi, K. Chakraborty, and S. Roy, "IcoNoClast: Tackling Voltage Noise in the NoC Power Supply Through Flow-Control and Routing Algorithms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 7, pp. 2035–2044, Jul. 2017.

[5] 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.

[6] 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.

[7] J. Boe, "Probabilistic Verification for Modular Network-on-Chip Systems," *All Graduate Theses and Dissertations, Spring 1920 to Summer 2023*, May 2023.

[8] 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.

[9] N. E. Jerger, T. Krishna, and L.-S. Peh, *On-Chip Networks*, ser. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2017.

[10] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Cham: Springer International Publishing, 2018.

[11] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, Oct. 1969.

[12] R. W. Floyd, "Assigning Meanings to Programs," in *Program Verification: Fundamental Issues in Computer Science*, T. R. Colburn, J. H. Fetzer, and T. L. Rankin, Eds. Dordrecht: Springer Netherlands, 1993, pp. 65–81.

[13] L. Lamport, "Composition: A Way to Make Proofs Harder," in *Compositionality: The Significant Difference*, G. Goos, J. Hartmanis, J. Van Leeuwen, W.-P. De Roever, H. Langmaack, and A. Pnueli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, vol. 1536, pp. 402–423.

[14] C. Baier and J.-P. Katoen, *Principles of Model Checking*, ser. The MIT Press Ser. Cambridge, Massachusetts London, England: The MIT Press, 2008.

[15] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[16] Y. Yu, P. Manolios, and L. Lamport, "Model Checking TLA+ Specifications," in *Correct Hardware Design and Verification Methods*, L. Pierre and T. Kropf, Eds. Berlin, Heidelberg: Springer, 1999, pp. 54–66.

[17] L. Lamport, "Specifying Concurrent Systems with TLA+," *Calculational System Design*, pp. 183–247, Apr. 1999.

[18] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer, 2004, pp. 168–176.

[19] T. Wu, X. Li, E. Manino, R. S. Menezes, M. R. Gadelha, S. Xiong, N. Tihanyi, P. Petoumenos, and L. C. Cordeiro, "ESBMC v7.7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Gurfinkel and M. Heule, Eds. Cham: Springer Nature Switzerland, 2025, pp. 223–228.

[20] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV version 2: An OpenSource tool for symbolic model checking," in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Copenhagen, Denmark: Springer, Jul. 2002.

[21] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker," in *Computer Aided Verification*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, A. Biere, and R. Bloem, Eds. Cham: Springer International Publishing, 2014, vol. 8559, pp. 334–342.

[22] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 134–152, Dec. 1997.

[23] 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*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer, 2014, pp. 593–598.

[24] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-Time Systems," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer, 2011, pp. 585–591.

[25] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, "The probabilistic model checker Storm," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 4, pp. 589–610, Aug. 2022.

[26] T. Spork, C. Baier, J.-P. Katoen, J. Piribauer, and T. Quatmann, "A Spectrum of Approximate Probabilistic Bisimulations," in *35th International Conference on Concurrency Theory (CONCUR 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Majumdar and A. Silva, Eds., vol. 311. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 37:1–37:19.

[27] H. Hermanns, B. Wachter, and L. Zhang, "Probabilistic CEGAR," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer, 2008, pp. 162–175.

[28] H. L. S. Younes and R. G. Simmons, "Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer, 2002, pp. 223–235.

[29] C. E. Budde, P. R. D'Argenio, A. Hartmanns, and S. Sedwards, "A Statistical Model Checker for Nondeterminism and Rare Events," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 340–358.

[30] A. Newell and H. Simon, "The logic theory machine–A complex information processing system," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 61–79, Sep. 1956.

[31] T. C. D. Team, "The Coq Proof Assistant," Zenodo, Sep. 2024.

[32] L. de Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds. Cham: Springer International Publishing, 2021, pp. 625–635.

[33] T. Nipkow, M. Wenzel, L. C. Paulson, G. Goos, J. Hartmanis, and J. Van Leeuwen, Eds., *Isabelle/HOL*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, vol. 2283.

[34] M. Kaufmann and J. Strother Moore, "ACL2: An industrial strength version of Nqthm," in *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, Jun. 1996, pp. 23–34.

[35] K. R. M. Leino and K. Leino, *Program Proofs*. Cambridge, Massachusetts London, England: The MIT Press, 2023.

[36] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6355, pp. 348–370.

[37] A. Zerdani and F. Boutekkouk, "An Overview of Formal Verification of Network-on-Chip (NoC) Methods," in *13th International Conference on Information Systems and Advanced Technologies "ICISAT 2023"*, M. R. Laouar, V. E. Balas, V. Piuri, D. Rad, Z. Touati Hamad, and A. Cheddad, Eds. Cham: Springer Nature Switzerland, 2024, pp. 26–34.

[38] A. Zaman and O. Hasan, "Formal verification of circuit-switched Network on chip (NoC) architectures using SPIN," in *2014 International Symposium on System-on-Chip (SoC)*. Tampere, Finland: IEEE, Oct. 2014, pp. 1–8.

[39] M. Dridi, M. Lallali, S. Rubini, F. Singhoff, and J.-P. Diguet, "Modeling and Validation of a Mixed-Criticality NoC Router Using the IF Language," in *Proceedings of the 10th International Workshop on Network on Chip Architectures*, ser. NoCArc '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1–6.

[40] J. C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, "CADP a protocol validation and verification toolbox," in *Computer Aided Verification*, G. Goos, J. Hartmanis, J. Leeuwen, R. Alur, and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, vol. 1102, pp. 437–440.

[41] 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, Mar. 2016.

[42] Y. Aydi, M. Baklouti, M. Abid, and J.-L. Dekeyser, "A multi–level design methodology of multistage interconnection network for MPSOCs," *International Journal of Computer Applications in Technology*, vol. 42, no. 2-3, pp. 191–203, Jan. 2011.

[43] A. Helmy, L. Pierre, and A. Jantsch, "Theorem proving techniques for the formal verification of NoC communications with non-minimal adaptive routing," in *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. Vienna: IEEE, Apr. 2010, pp. 221–224.

[44] F. Verbeek and J. Schmaltz, "On Necessary and Sufficient Conditions for Deadlock-Free Routing in Wormhole Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 12, pp. 2022–2032, Dec. 2011.

[45] ——, "Proof Pearl: A Formal Proof of Dally and Seitz' Necessary and Sufficient Condition for Deadlock-Free Routing in Interconnection Networks," *Journal of Automated Reasoning*, vol. 48, no. 4, pp. 419–439, Apr. 2012.

[46] K. L. McMillan and O. Padon, "Ivy: A Multi-modal Verification Tool for Distributed Algorithms," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 190–202.

[47] A. E. Kiasari, Z. Lu, and A. Jantsch, "An Analytical Latency Model for Networks-on-Chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 113–123, Jan. 2013.

[48] M. Slijepcevic, M. Fernadez, C. Hernandez, J. Abella, E. Quiñones, and F. J. Cazorla, "pTNoC: Probabilistically Time-Analyzable Tree-Based NoC for Mixed-Criticality Systems," in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug. 2016, pp. 404–412.

[49] 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)*, Mar. 2019, pp. 634–637.

[50] Z. Sharifi, S. Mohammadi, and M. Sirjani, "Comparison of noc routing algorithms using formal methods," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer …, 2013, p. 461.

[51] K. Tatas, "Towards an Analytical Model of Latency in Deflection Routing: A Stochastic Process Approach for Bufferless NoCs," in *2021 10th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, Jul. 2021, pp. 1–5.

[52] M. L. Littman, S. M. Majercik, and T. Pitassi, "Stochastic Boolean Satisfiability," *Journal of Automated Reasoning*, vol. 27, no. 3, pp. 251–296, Oct. 2001.

[53] M. Samer and S. Szeider, "Algorithms for propositional model counting," *Journal of Discrete Algorithms*, vol. 8, no. 1, pp. 50–64, Mar. 2010.

[54] N.-Z. Lee and J.-H. R. Jiang, "Towards Formal Evaluation and Verification of Probabilistic Design," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1202–1216, Aug. 2018.

[55] A. Pnueli, S. Ruah, and L. Zuck, "Automatic Deductive Verification with Invisible Invariants," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and W. Yi, Eds. Berlin, Heidelberg: Springer, 2001, pp. 82–97.

[56] P. A. Abdulla, F. Haziza, and L. Holík, "All for the Price of Few," in *Verification, Model Checking, and Abstract Interpretation*, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds. Berlin, Heidelberg: Springer, 2013, pp. 476–495.

[57] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan. 1967.

[58] K. L. McMillan, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *Computer Aided Verification*, G. Goos, J. Hartmanis, J. Van Leeuwen, A. J. Hu, and M. Y. Vardi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, vol. 1427, pp. 110–121.

[59] T. Henzinger, X. Liu, S. Qadeer, and S. Rajamani, "Formal specification and verification of a dataflow processor array," in *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, Nov. 1999, pp. 494–499.

[60] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Request for Comments RFC 9000, May 2021.

[61] K. L. McMillan and L. D. Zuck, "Compositional Testing of Internet Protocols," in *2019 IEEE Cybersecurity Development (SecDev)*. Tysons Corner, VA, USA: IEEE, Sep. 2019, pp. 161–174.

[62] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-Guarantee Verification for Probabilistic Systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds. Berlin, Heidelberg: Springer, 2010, pp. 23–37.

[63] P. Nuzzo, J. Li, A. L. Sangiovanni-Vincentelli, Y. Xi, and D. Li, "Stochastic Assume-Guarantee Contracts for Cyber-Physical System Design," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 1, pp. 1–26, Jan. 2019.

[64] R. Calinescu, S. Kikuchi, and K. Johnson, "Compositional Reverification of Probabilistic Safety Properties for Large-Scale Complex IT Systems," in *Large-Scale Complex IT Systems. Development, Operation and Management*, R. Calinescu and D. Garlan, Eds. Berlin, Heidelberg: Springer, 2012, pp. 303–329.

[65] K. R. M. Leino, "Modeling Concurrency in Dafny," in *Engineering Trustworthy Software Systems*, J. P. Bowen, Z. Liu, and Z. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 115–142.

[66] C. Métayer, J.-R. Abrial, and L. Voisin, "Event-B Language," May 2005.

[67] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, 2018.

[68] *IEEE Standard for VHDL Language Reference Manual*, 2019.

[69] N. Waddoups, "Artifact for "modular verification for network-on-chip designs using probabilistic model checking and assume-guarantee reasoning"," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17822035

[70] H. Bohnenkamp, P. D'Argenio, H. Hermanns, and J.-P. Katoen, "MODEST: A Compositional Modeling Formalism for Hard and Softly Timed Systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 812–830, Oct. 2006.

[71] M. A. Malikov and A. Y. Romanov, "Traffic Patterns in Networks-on-Chip: A Survey," *IEEE Access*, vol. 13, pp. 148 803–148 815, 2025.

APPENDICES

APPENDIX A

Proofs

A.1 Equivalence of Modest Lists and Dafny Sequence Operations

To demonstrate equivalence between these following operations on a list data structure in MODEST and a sequence in Dafny we first translated the list type and related functions into Dafny. Then we detailed our goal for proving correctness between the two types and encode that proof into Dafny.

Datatypes

The definition of the buffer type (as a linked list) in MODEST is defined as a `datatype` (a struct) with two members: `hd` holds the packet, and `tl` optionally holds another buffer instance. This recursively creates a linked list structure.

Code Segment A.1. MODEST Buffer

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

// List of length two
buffer option ll2 = buffer { 0, buffer { 1, none }};
```

In Dafny, this is idiomatically encoded as an algebraic datatype with two options, `hd` and `tl`. The list is created here by appending together several instances of this datatype.

To represent the bounded integer `int(0..NOC_MAX_ID`, we created a bounded integer type, `packet`, and used it in both the list-based and sequence datatypes. There are two datatypes shown in Dafny, `buffer_ll` and `buffer_s`, which represent our list based buffer and sequence based buffer, respectively.

Code Segment A.2. Dafny Buffer

```
const NOC_MAX_ID: nat

type id_t = i: nat | i ⩽ NOC_MAX_ID

datatype buffer_ll =
| hd(id: id_t, next: buffer_ll)
| tl

type buffer_s = seq<id_t>
```

The following definition encodes what it means for a linked list buffer to be equal to the sequence based buffer.

Definition A.1. A linked list buffer `ll: buffer_ll` and a sequence buffer `s: buffer_s` are equal iff for each node in `ll` there is a corresponding element in `s`.

```
predicate equal(ll: buffer_ll, s: buffer_s)
  decreases ll, s
{
  if ll.tl? && |s| == 0 then true
  else if ll.hd? && |s| > 0 then ll.id == s[0] && equal(ll.next, s[1..])
  else false
}
```

For these two datatypes we aim to prove equivalence between the following actions.

Table A.1: Semantic Mapping of Buffer Operations

Operation	**List**	**Sequence**
Enqueue	`enqueue(n, b)`	`b + [n]`
Dequeue	`dequeue(b)`	`b[1..]`
Peek	`peekFront(b)`	`b[0]`
Length	`len(b)`	`|b|`
Containment	`contains(n, b)`	`n in b`

The proofs, encoded in Dafny, are shown in Code Segment A.3.

Code Segment A.3. Linked list and sequence equal for given semantics

```
module List {
  const NOC_MAX_ID: nat

  type packet = i: nat | i ⩽ NOC_MAX_ID

  datatype buffer_ll =
  | hd(p: packet, next: buffer_ll)
  | tl

  type buffer_s = seq<packet>

  predicate equal(ll: buffer_ll, s: buffer_s)
    decreases ll, s
  {
    if ll.tl? && |s| == 0 then true
    else if ll.hd? && |s| > 0 then ll.p == s[0] && equal(ll.next, s[1..])
    else false
  }

  function init(): buffer_ll {
    tl
  }

  lemma initEqual(ll: buffer_ll, s: buffer_s)
    requires ll == init() && s == []
    ensures equal(ll, s)
  {}

  function len(ll: buffer_ll): nat {
    if ll.tl? then 0 else 1 + len(ll.next)
  }

  lemma lenEqual(ll: buffer_ll, s: buffer_s)
    requires equal(ll, s)
    ensures len(ll) == |s|
  {}

  function enqueue(p: packet, ll: buffer_ll): buffer_ll {
    hd(p, ll)
  }
```

```
lemma enqueueEqual(ll: buffer_ll, s: buffer_s, p: packet)
 requires equal(ll, s)
 ensures equal(enqueue(p, ll), [p] + s)
{}

function dequeue(ll: buffer_ll): buffer_ll {
 match ll
 case tl ⇒ tl
 case hd(p, n) ⇒
  if n.tl? then tl
  else hd(p, dequeue(n))
}

lemma dequeueEqual(ll: buffer_ll, s: buffer_s)
 requires equal(ll, s) && len(ll) > 0
 ensures equal(dequeue(ll), s[..|s| - 1])
{
 if |s| ≠ 0 && ll.hd? {
  var s' := s[1..];
  var ll' := ll.next;

  if |s'| == 0 && ll'.tl? {
   // automatic
  } else {
   dequeueEqual(ll', s');
   assert equal(dequeue(ll'), s'[..|s'| - 1]);
   assert dequeue(ll) == hd(ll.p, dequeue(ll'));
   assert s[..|s| - 1] == [s[0]] + s'[..|s'| - 1];
  }
 } else {
  // automatic
 }
}

predicate contains(p: packet, ll: buffer_ll) {
 if ll.tl? then false
 else if ll.p == p then true
 else contains(p, ll.next)
}

lemma containsEqual(ll: buffer_ll, s: buffer_s, p: packet)
 requires equal(ll, s)
 ensures contains(p, ll) ⟺ p in s
{}
```

```
  function peekFront(ll: buffer_ll): packet
    requires len(ll) > 0
  {
    if ll.next.tl? then ll.p
    else peekFront(ll.next)
  }

  lemma peekFrontEqual(ll: buffer_ll, s: buffer_s)
    requires equal(ll, s) && len(ll) > 0
    ensures peekFront(ll) == s[|s| - 1]
  {}
}
```