VERIFICATION METHODOLOGIES FOR FAULT-TOLERANT NETWORK-ON-CHIP SYSTEMS

by

Zhen Zhang

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

May 2016

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Zhen Zhang**

has been approved by the following supervisory committee members:

Chris J. Myers, Chair		**October 26, 2015** Date Approved
Wendelin Serwe, Member		**October 26, 2015** Date Approved
Hao Zheng, Member		**October 26, 2015** Date Approved
Kenneth S. Stevens, Member		**October 26, 2015** Date Approved
Priyank Kalla, Member		**October 26, 2015** Date Approved

and by **Gianluca Lazzi**, Chair of

the Department of **Electrical and Computer Engineering**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Over the last decade, *cyber-physical systems* (CPSs) have seen significant applications in many safety-critical areas, such as autonomous automotive systems, automatic pilot avionics, wireless sensor networks, etc. A CPS uses networked embedded computers to monitor and control physical processes. The motivating example for this dissertation is the use of fault-tolerant routing protocol for a *Network-on-Chip* (NoC) architecture that connects *electronic control units* (ECUs) to regulate sensors and actuators in a vehicle. With a network allowing ECUs to communicate with each other, it is possible for them to share processing power to improve performance. In addition, networked ECUs enable flexible mapping to physical processes (e.g., sensors, actuators), which increases resilience to ECU failures by reassigning physical processes to spare ECUs. For the on-chip routing protocol, the ability to tolerate network faults is important for hardware reconfiguration to maintain the normal operation of a system. Adding a fault-tolerance feature in a routing protocol, however, increases its design complexity, making it prone to many functional problems. Formal verification techniques are therefore needed to verify its correctness.

This dissertation proposes a link-fault-tolerant, multiflit wormhole routing algorithm, and its formal modeling and verification using two different methodologies. An improvement upon the previously published fault-tolerant routing algorithm, a link-fault routing algorithm is proposed to relax the unrealistic node-fault assumptions of these algorithms, while avoiding deadlock conservatively by appropriately dropping network packets. This routing algorithm, together with its routing architecture, is then modeled in a process-algebra language LNT, and compositional verification techniques are used to verify its key functional properties. As a comparison, it is modeled using channel-level VHDL which is compiled to *labeled Petri-nets* (LPNs). Algorithms for a partial order reduction method on LPNs are given. An optimal result is obtained from heuristics that trace back on LPNs to find causally related enabled predecessor transitions. Key observations are made from the comparison between these two verification methodologies.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to thank my PhD advisor, Chris Myers. Chris introduced me to the world of formal verification after I joined his research group, and has been patiently guiding me to explore new knowledge in this field throughout my Ph.D. His guidance has significantly helped me to acquire necessary skills to become a fine scholar. He taught me many things, including academic paper writing, research presentation, and conducting paper reviews. Most importantly, he taught me practical principles of analyzing a given complex engineering problem: always start with a small and simplified version, and gradually build up the understanding of the whole problem. I will forever be grateful to him for assisting me in achieving my PhD.

I would not be the person I am today if it were not for my family. My wonderful parents, Ling Zhu and Chunning Zhang, have been a great source of unconditional and continuous love and support throughout my life. I will be forever indebted to their great effort in providing me the best education opportunities.

I would also like to thank my lab mates, Curtis Madsen, Nicholas Roehner, Andrew Fisher, Leandro Watanabe, and Tramy Nguyen. They each helped me to really become part of the group. Curtis and Nic generously invited me to dinner parties on major American holidays, provided an outlet to discuss frustrations, and hosted many fun board game nights. Curtis was also a great companion for discussions on my research work. Nic shared many of his thoughts with me on diverse topics from philosophy, music, and science fiction. Andrew taught me numerous topics on mathematical foundations related to my own work and helped me to get my thoughts straight. Leandro and Tramy provided discussions on topics of good software practice for my Synthetic Biology Open Language Java library project. In addition, I would like to thank the rest of my lab mates, Dhanashree Kulkarni, Robert Thacker, Satish Batchu, and Kevin Jones for their help on the Lema tool; and Xiaojun Sun, Jinpeng Lv, and Tim Pruss for conversations on computer algebra.

I would like to thank my committee, Wendelin Serwe, Hao Zheng, Ken Stevens, and Priyank Kalla. Wendelin taught me process algebra and has been a source of inspiration on many intellectually challenging problems. Hao exposed to me many verification techniques

that also inspired my own research work. Ken and Priyank provided valuable feedback on my thesis and ideas for some future work.

Finally, I would like to thank all the past members of the lab who contributed to LEMA. Without their work, this work would not be possible. In particular, I would like to thank Satish Batchu, Kevin Jones, Dhanashree Kulkarni, Robert Thacker, Scott Little, and David Walter for their contributions to LEMA.

This material is based upon work supported by the National Science Foundation under grants CNS-0930510 and CNS-0930225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Part of this work was performed during my visit at the Inria Grenoble-Rhône-Alpes research centre.

CHAPTER 1

INTRODUCTION

A *cyber-physical system* (CPS) is characterized by the tight interaction between a digital computing component (the cyber part) and a continuous-time dynamical system (the physical part). A CPS uses networked embedded computers to monitor the physical processes through feedback loops and issues adjustment control signals to them accordingly [1]. CPSs nowadays have ubiquitous applications in many areas, such as avionics, advanced automotive systems, robust medical devices, etc. One active area of CPS application is in the automotive industry. Modern vehicles can have up to 80 *electronic control units* (ECUs) that control and operate everything from the engine and brakes to door locks and electric windows. Currently, each ECU is statically tied to its specific sensors and actuators. This means that processing power between different ECUs cannot be shared, which may degrade the performance of the chip due to imbalanced workload on each ECU. More importantly, this structure is susceptible to faults as if an ECU fails, it causes a malfunction in the corresponding sensor/actuator. With the advances in semiconductor technology, it is now possible to have multiple cores on a single chip which communicate using a *Network-on-Chip* (NOC) paradigm. A NOC approach allows flexible mapping between ECUs and sensors/actuators, which makes it possible for ECUs to share processing power and tolerate faults by having spare units. Designing a fully functional NOC system is challenging. Specifically, the routing algorithm implemented on the NOC structure has to be fault-tolerant and guarantee deadlock freedom. Adding fault-tolerance adaptivity to a routing algorithm increases its design complexity and makes it prone to deadlock and other problems if improperly implemented.

1.1 Model Checking

To guarantee functional correctness of a complex NOC routing algorithm, formal verification techniques are needed to reason about its concurrent behavior. *Model checking* is an automated technique for the verification of finite-state systems. It involves three tasks [2]: modeling, specification, and verification. A mathematical model of the system of interest is constructed using some formalism, such as *labeled transition systems*, *Petri nets*,

Büchi automata, etc. Model construction can be automated as a compilation task from a high-level language to the language specified by the formalism. The next step is to specify properties that the system must satisfy. *Temporal logic* has been widely used as a formalism for this purpose, due to its ability to specify the system behavior over time. The last step is to automatically build the *state space* of the system model, which means exhaustively enumerating all states that are reachable by the model and all transitions in which the system evolves from one state to another. The properties can be checked during or after the state space construction of a model. If a violation of a property is identified, the model checker usually terminates and reports a *counterexample* to the user. A counterexample is an error trace consisting of a sequence of states and transitions that start from the initial state to the state with the property violation. Many model checkers are capable of providing accurate and shortest error traces to the user for debugging purposes. One advantage of model checking is that it automates many verification tasks and does not rely much on user's special skills. The only places where human assistance is needed are specifying the properties and analyzing the verification results. Also, many different kinds of properties can be verified from the constructed state space of the system model [2, 3].

Model checking has been successfully applied to the verification of a variety of hardware and software systems over the past few decades. It, however, suffers from the fundamental difficulty of the *state explosion* problem for large-scale designs. This problem occurs when the generated states during model checking become too large to fit into the computer memory. Roughly speaking, the size of a state space grows exponentially with the number of processes and variables in a system [2]. For example, for a system with n concurrent processes, each of them having α states, then the state space for that system can be as large as α^n. Many techniques have been developed in recent years to address the state explosion problem. *Symbolic model checking* [4, 5] has proven to be a powerful technique to deal with the state space problem. Specifically, symbolic state representation such as *ordered binary decision diagrams* (OBDDs) [6] has provided a compact form for Boolean formulas. Efficient algorithms [7] have been developed based on OBDDs to verify designs with extremely large state spaces. *Bounded model checking* (BMC) [8] is another flavor of symbolic model checking that is based on *propositional decision procedures* (SAT) [9]. The basic idea of BMC is to generate a propositional formula from a counterexample of a bounded length, and check the propositional formula with a SAT solver. A satisfiable formula reported by the SAT solver represents a concrete counterexample showing the property violation. Otherwise, the bound is increased and the process repeats. Complete extensions to BMC allow one to stop this

process at some point, with the conclusion that the property cannot be violated, hopefully before the available resources are exhausted. *Compositional verification* techniques address the state explosion problem from a different perspective. They either avoid generating the global state space by performing local analysis on each component's state space, e.g. [10–12], or iteratively construct and minimize the local state space for a component and compose it with other components to gradually form the state space for the whole system, e.g. [13, 14]. *Abstraction* [15, 16] computes an overapproximation of the original model by building a small set of data values from the actual data values in the system based on a specified mapping, and the resultant smaller system is used to verify properties at an abstract level with less complexity [2]. *Symmetry reduction* [17–19] exploits the symmetry relation to construct equivalence class representatives in the model, and limits the state search to them in order to save memory and runtime. *Partial order reduction* (POR) [20–25], has been proven to successfully reduce the state space for systems with concurrency, such as most asynchronous systems. It exploits the commutativity of concurrently executed transitions, which reach the same state when executed in a different order. Only one such order is selected and executed, and thus the resultant state space is significantly smaller than that generated by exploring all possible orders. There has been research work on combining the said state reduction methods for better reductions. A combination of POR with symmetry reduction in [26] shows a much smaller reduced state space than that obtained from applying either method individually. Valmari [27] proposed a similar idea for deadlock detection for colored Petri-nets. POR techniques have also been applied to symbolic BDD-based invariant checking [28]. An approach that combines compositional analysis with POR [29] extracts dependency information from each component of a system and forms compositional rules for global dependency information. Offline POR has also been combined with BMC techniques to show that threshold-based distributed algorithms have a similar execution of bounded length [30, 31].

1.2 Contributions

Formal verification techniques are therefore needed to verify its correctness. This dissertation proposes a link-fault-tolerant, multiflit wormhole routing algorithm, and its formal modeling and verification using two different methodologies.

Improving upon Glass and Ni's routing algorithm [32] that assumes node faults, this dissertation proposes a routing architecture extending that introduced by Wu et al. [33] to a multiflit wormhole setting. It loosens Glass and Ni's impractical assumption to achieve

link-fault tolerance, covering a wider range of link fault cases that Glass and Ni's algorithm fails to handle. Deadlock avoidance is implemented conservatively with adequate packet drops to break the cycle of dependencies. Simulation results indicate that this algorithm provides significant improvements in network reliability with minimal cost.

This link-fault routing algorithm is modeled in the process-algebraic language LNT [34]. With the help of the CADP verification tool box, formal analysis exposes design flaws leading to false behaviors such as a packet leakage path leading to unintended packet drop and deadlock caused by removing arbiter's buffering capacity. To combat the notorious state explosion problem, a data abstraction technique [35] is applied to map the destination coordinates of a packet to a Boolean value representing its diversion status. Mismatch between the abstract and concrete models leads to the discovery of a potential livelock problem due to redundant packet diversions. Elimination of these diversions leads to an improved algorithm that simplifies the routing architecture, enabling successful compositional verification. The routing algorithm is proven to have several desirable properties, including deadlock and livelock freedom, and tolerance to a single-link-fault [36].

As a comparison, the derived livelock-free routing protocol is modeled using the channel-level VHDL that is automatically compiled to *labeled Petri-nets* (LPNs) [37, 38] for verification using the LEMA tool. Algorithms are described for an ample set-based *partial order reduction* (POR) technique, which analyzes transition dependencies through a recursive trace-back search on LPNs. A set with the least number of enabled transitions that need interleaving is selected at each state. Cost and benefit of using trace-back are evaluated on several nontrivial asynchronous circuit models, and are compared to LNT models on a series of buffers that uses asynchronous communication. Although POR achieves significant state reduction on certain arbiter models, it is still outperformed by composition minimization of CADP on the corresponding LNT models. Root cause of the difference is analyzed by comparing the LNT and LPN specifications.

1.3 Dissertation Outline

This dissertation is organized as follows. Syntax and semantics of the two modeling formalisms, namely LNT and LPN, are presented in Chapter 2. Examples of asynchronous channel models are presented in LNT first, and correspondingly, in the channel-level VHDL, which is then automatically compiled to LPNs. Common features and differences of these models are described in this chapter.

Chapter 3 describes Glass and Ni's fault-tolerant routing algorithm on a two-dimensional mesh network, and demonstrates, with examples, how this and other similar routing algo-

rithms fail to handle link-faults. This chapter then describes the proposed link-fault-tolerant routing algorithm and its architectural design. The deadlock avoidance mechanism is explained in detail with some illustrative examples.

The LNT specification of the link-fault model is presented in Chapter 4. Lessons learned during the design process are described, followed by a description of the data abstraction that enables verification of deadlock freedom and single-link-fault tolerance. This chapter then describes the discovery of a potential livelock problem. In the process of eliminating this problem, an improved routing architecture is derived. The improvement simplifies the routing architecture, enabling successful verification using the CADP verification toolbox. The routing algorithm is proven to have several desirable properties, including deadlock and livelock freedom, and tolerance to a single-link-fault.

Chapter 5 describes an alternative way of modeling and verification of the same link-fault-tolerant NoC routing protocol using the LEMA tool. It presents in detail a partial order reduction method on LPNs with trace-back that optimally reduces unnecessary transition interleavings for state reachability analysis, avoiding exploring unimportant state-transition sequences. The cost and benefits of using trace-back are evaluated on a series of asynchronous circuits examples. Comparisons of state reduction and performances are drawn between LNT and LPN on a series of asynchronous buffer examples. VHDL models for the representative routers and arbiters are described for the livelock-free link-fault-tolerant routing protocol that is presented in the pervious chapter. They are automatically compiled to LPNs which are used for state exploration by LEMA. Although partial order reduction manages to achieve significant state reduction on certain arbiter models, it is still outperformed by composition minimization of CADP on the corresponding LNT models. Key observations are made on the comparison of the LNT and LPN specifications.

Finally, Chapter 6 concludes this dissertation by giving a summary of the work and by presenting future research directions.

CHAPTER 2

MODELS OF CONCURRENT SYSTEMS

This chapter introduces the underlying model constructs for the behavioral modeling of the NoC routing architecture in subsequent chapters. It starts with a brief introduction of major process algebraic approaches in providing formal syntax and semantics of concurrent systems. The rest of the chapter focuses on describing key modeling constructs in terms of two approaches: the process algebraic language LNT, and the channel-level VHDL that can be automatically compiled to the LPN modeling formalism. Both modeling languages are introduced first with their respective verification environment, followed by a detailed description of communication channels, nondeterministic choice, parallel composition, and a subset of the control structures that are relevant to the modeling of NoC routing architectures.

2.1 A Brief Introduction to Process Algebra

Over the last forty years, process algebras have witnessed significant success in providing formal semantics of concurrent systems towards verification and validation. Process algebras are mathematical models of processes that are abstractions of components of a system that continuously interact with each other and with their common environment. A process algebra often provides process terms that are generated from its abstract syntax for specifying components of a system, and an operational semantics that associates each term with a *Labeled Transition System* (LTS) that consists of a set of states, a transition relation, and a set of transition labels. The three main approaches in providing semantics of syntactically correct process terms are operational, denotational, and algebraic semantics, with the operational semantics playing a central role in all process algebras. Moreover, a process algebra usually includes mechanisms for observing behavioral equivalences between two systems or between an abstract and a concrete one.

At its core, the abstract syntax of a process algebra defines operators in that language. It consists of combinations of elementary terms and operators, both of which are the ingredients for building basic process terms. Starting with a set of basic processes and actions, one can build new processes. Common operators in almost all known process algebras include

nondeterministic and parallel compositions of processes; the abstraction operation such as the hiding, restricting, and renaming operators that limit the interface of a process; and modeling infinite behaviors from finite operations.

An *operational semantics* provides an abstract machine-based view of computation by treating a program as a LTS. Based on structural induction offered by the *Structural Operational Semantic* (SOS) [39], each process term (or operator) is specified by a set of inference rules that describe its behavior through the behaviors of its composing components, each of which has its own process term that is specified by the corresponding set of inference rules. In fact, each process term can be considered as a component that can interact with other components or with its environment. Regarding its association with a LTS, process algebra terms are represented as states in their corresponding LTSs; actions are described by a transition relation going from a given state to its next possible one; and the visibility of each action is denoted by its corresponding transition label. With features from structural induction, the LTS of a complex system can be composed from its component LTSs. A *denotational semantics* establishes a mapping from a language to an abstract model so that the meaning of a program is determined by the meaning of each of its immediate subcomponents. Ideally, the abstract model should be able to describe the "essence" of programs in the language that is mapped to it. Semantic clauses for different operators can then be devised in order to specify semantic properties of a program that consists of these operators. An *algebraic semantics* is defined by a set of algebraic laws that are basic axioms of an equational system.

Behavioral equivalences are useful in proving whether two systems are equivalent or how one system approximates the other. Specifically, during an incremental process of constructing a system's LTS, one can replace a subcomponent by its behaviorally equivalent abstract counterpart without affecting the overall behavior of the system. Built in the definitions of different behavioral equivalences are aspects of behaviors that can or cannot be ignored. It is, therefore, necessary to know the properties that a behavioral equivalence preserves, as process equivalence is defined in terms of what equalities can be proved using them. Families of major behavioral equivalences are reviewed in Chapter 4.

The process algebra *Calculus of Communicating Systems* (CCS) [40, 41] was introduced by Robin Milner around 1980. Actions in CCS model atomic, two-way communications, i.e., nondivisible communications between exactly two participants, and common operators such as (binary) parallel composition, nondeterministic selection, restriction, and relabeling operations, etc. The basic semantics of CCS is operational where each CCS process term is

associated with a LTS. This allows the development of theories for its behavioral equivalences to be based on LTSs. A successor of CCS, *π-calculus* [42], was developed by Milner et al. to target the description of concurrent systems with dynamically changing configurations during the computation.

Another well-known process algebra is the *Communicating Sequential Processes* (CSP) invented by C.A.R. Hoare [43, 44]. It was designed with the goal of providing a notation and theory for the analysis of different components of a system interacting with each other through communication, and has served as a mechanism for understanding and applying concurrency theory extensively over the years. Unlike CCS whose original design was provided with operational semantics, CSP has been given a number of denotational semantics, mainly based on sets of behaviors such as traces, failures, and divergences, which are used for deciding process equivalence. The operational semantics of CSP, heavily influenced by the work on CCS, was historically developed to provide an alternative view in addition to its denotational semantics. Many tools that support CSP both for teaching and industrial applications have emerged over the years, and have contributed significantly to the formal description and (automated) analysis of industrial-sized problems. A comprehensive text on the fundamental theories and applications of CSP can be found in [45]. It is worth mentioning that the maturity of CSP in modeling communication and concurrency lent itself to many other domain-specific languages. For instance, the *Communicating Hardware Processes* (CHP) by Martin [46], and the *Tangram* language by van Berkel et al. [47], were adapted from CSP for modeling and designing asynchronous VLSI systems.

Taking from a more abstract and completely different viewpoint of process algebra, the *Algebra of Communicating Processes* (ACP) [48] provides a purely algebraic approach to concurrency theory. ACP takes process algebra as a mathematical structure, which consists of a set of processes and a set of operators such as sequential, nondeterministic, or parallel composition, communication, etc. All operations satisfy conditions governed by a set of axioms that are usually presented as a set of formal equations. Operational semantics and behavioral equivalences are possible models over which the algebra can be defined and the axioms can be applied.

The *Language Of Temporal Ordering Specification* (LOTOS) is yet another process algebra that was developed and standardized within International Standards Organization for specifying and verifying communication protocols during the years 1981-1988 [49]. The goal was to provide an unambiguous, precise, and complete formal description language with a well-defined basis for the verification and validation of the *Open Systems Interconnection*

(OSI) telecommunication standards. Borrowing features from both CSP and CCS, LOTOS handles process behaviors that are common to both process algebras. It uses gates that correspond to CSP channels for specifying communication, and extends this feature to do multiway synchronization that is not in either CCS or CSP; and the nondeterministic choice in LOTOS follows the CCS style with an internal event, but its parallel composition follows the CSP approach [50]. Additionally, it is able to describe data structures and value expressions using abstract data type technique ACT-ONE [51].

2.2 Behavioral Modeling and Verification in CADP

The *Construction and Analysis of Distributed Processes*[1] (CADP) toolbox has been employed in many industrial projects for the design and analysis of asynchronous concurrent systems. More than 150 case-studies have been published, covering a wide range of applications, such as shared-memory mutual exclusion protocols [52], mobile ad hoc networks [53], dynamic management protocol for cloud applications [54], and logical regulatory modules for intercellular networks in Biology [55]. The modeling formalism is the LNT language, a process algebraic formal specification language that has been developed and implemented in the CADP toolbox since 2005. The LNT language has its roots in the LOTOS language. The LOTOS language, however, has limited data types that do not meet the users' needs and cannot handle real-time constraints. Extensions with significant improvements to the LOTOS language have been made and included in the LNT language, such as fully imperative syntax and semantics, the *Rich Term Syntax* notations, etc. The Rich Term Syntax notations allow complicated data types to be expressed naturally in LNT, compared to the LOTOS abstract data types. One example is the limited range of integers, i.e., integers between 0 and 9, in the LOTOS `NATURAL` library. To express numbers greater than 9, one has to use `Succ`. For instance, the natural number 12 in LNT is expressed as `Succ( Succ( Succ (9)))` in LOTOS. The Rich Term Syntax also provides standard notations for literal constants (e.g., lists, sets, strings) in LNT. Combining features from process calculi, and both functional and imperative languages, the concise and expressive power of the LNT language gives it many advantages in formally specifying complex concurrent systems.

Figure 2.1 shows a subset of tools from the CADP toolbox for model checking of a LNT model. The input LNT model is translated into a LOTOS specification in two steps: the LPP tool first expands the Rich Term Syntax notations in the LNT model into lower-level algebraic terms that are compatible with the LOTOS syntax, after which the LNT2LOTOS

[1] `http://cadp.inria.fr`

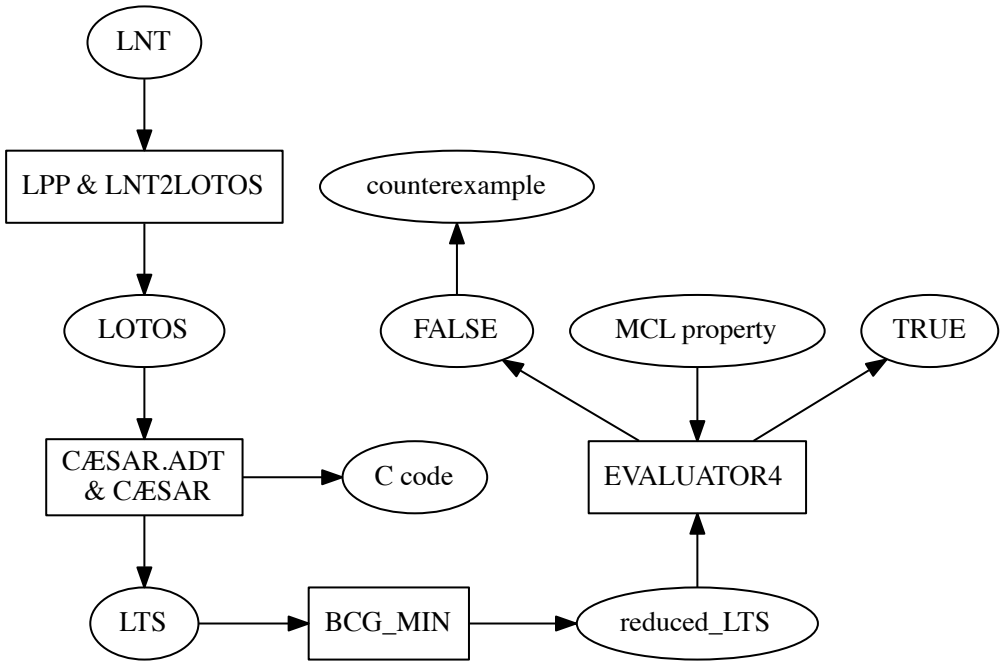


Figure 2.1: CADP tool flow.

tool translates the preprocessed LNT specification into a complete LOTOS specification. It is then compiled by the abstract data type compiler CÆSAR.ADT, which translates the data part of the LOTOS specification into C types and functions. They are used by the CÆSAR tool to compile the behavioral part of the LOTOS specification into either a C program or a *Labeled Transition System* (LTS). The C program can be executed and simulated, as well as embedded in real applications to allow rapid prototyping. The LTS can be used for equivalence checking and/or model checking with temporal logic properties. Stored as the *Binary Coded Graph* (BCG) format, a LTS can be minimized by the BCG_MIN tool according to strong, branching, or divergence-sensitive branching bisimulation relations. To model check a temporal logic property expressed as *Model Checking Language* (MCL) formulas, the EVALUATOR4 tool performs on-the-fly verification on the given LTS, and produces the verification results (**true** or **false**). A **false** result may optionally be accompanied by a counterexample in the form of a transition sequence or an LTS. CADP features the *Script Verification Language* (SVL) [56], which automates invocations of all aforementioned tools.

2.2.1 LNT Module

A module in the LNT language is the basic building block for specifying models of a system. It is formally defined as

$$
\begin{aligned}
\textit{lnt\_file} \quad ::= \quad & \textbf{module}\ M[(M_0,\ \ldots,\ M_m)] \\
& \quad [\textbf{with}\ \textit{predefined\_function}_0,\ \ldots,\ \textit{predefined\_function}_n]\ \textbf{is} \\
& \quad \textit{module\_pragma}_1\ \ldots\ \textit{module\_pragma}_p \\
& \quad \textit{definition}_0\ \ldots\ \textit{definition}_q \\
& \textbf{end module}
\end{aligned}
$$

where $(M_0, ..., M_m)$ is a set of imported module identifiers. All definitions in the imported modules are visible to module M and can be used by definitions in this module. Predefined functions defined by the `LNT_V1.lib` library provide data operations over both the six basic types, i.e., Booleans, natural numbers (`Nat`), integers (`Int`), real numbers (`Real`), characters (`Char`), and strings (`String`), as well as nonbasic types, e.g., list, sorted list, set. Details of these functions are described in Chapter 5 and Appendix C of [34]. Module pragmas can modify the default ranges of the three predefined types: natural numbers, integers, and strings. For example, "!*nat_bits* 3" limits the number of bits to 3 for all variables of type `Nat`, and "!*string_card* 5" limits the maximal cardinality of `String` to 5.

Each definition in a module defines one of the four entities: *type*, *function*, *channel*, or *process*. The type and function definitions allow specifications of customized data types and their operations, and the channel definition allows one to specify a set of gate profiles with custom gate types. These entities are described here informally with examples, and their grammar and semantics are defined in [34]. The process definition is given detailed explanations in the following sections.

In the following example, a data type NodeRange is defined as a closed range $[0, 4]$ of natural numbers, and Coordinates is defined as a pair of natural numbers, each of type NodeRange. The Coordinates type declares predefined functions get to retrieve its x and y values, and the infix function "!=" to do inequality check based on structural equivalence between two values of type Coordinates. The infix function "+" defines the addition operation of two values typed NodeRange as being added as natural numbers, and returns the result in NodeRange type.

```
module datatypes is
type NodeRange is
   range 0 .. 4 of Nat
end type

type Coordinates is
   Coordinates(x: NodeRange, y: NodeRange)
   with "get", "!="
end type

function _+_ (x, y: NodeRange) : NodeRange is
    return NodeRange (Nat (x) + Nat (y))
end function end module
```

A channel is defined as a set of gate profiles, each of which defines a list of gate types. Consider the type T and channel C, defined as follows:

```
type T is
Request, Response
end type
channel C is (T) end channel
```

The channel C includes two gates, namely "Request" and "Response". Two gates are considered as having compatible type, if and only if both are polymorphic or are declared with the same channel identifier. A channel is similar to the VHDL port declaration described in Section 2.3.1.

2.2.2 LNT Process

A system's behavior can be described using LNT processes, formally defined as follows:

process_definition ::= **process** Π [[*gate_declaration*$_0$,...,*gate_declaration*$_m$]]
[(*formal_parameters*$_1$,...,*formal_parameters*$_n$)]
[**raises** *exception_declaration*$_0$,...,*exception_declaration*$_k$] **is**
process_pragma$_1$...*process_pragma*$_l$
B
end process

where the process has a unique identifier Π, and can optionally be parameterized by a list of formal gates, a list of formal variables, and a list of formal exceptions. The optional list of process pragmas provides instructions for translating LNT source code to LOTOS and C. The process body B describes the behavior.

Gate declarations specify all communicating gates that are visible to other processes. Formal gate parameters must declare their types, which can be either a channel Γ or a polymorphic type using the **any** keyword:

$$
\begin{aligned}
\textit{gate\_declaration} \quad ::= \quad & G_0, \ldots, G_n : \Gamma \\
| \quad & G_0, \ldots, G_n : \textbf{any}
\end{aligned}
$$

Each of the formal parameters in the process definition can be declared with one of the three modes: the default mode *in* denotes a constant parameter whose value can be changed locally by a process and the change remains invisible to other processes; the *out* mode requires a parameter to be assigned to a value locally, and the value is visible outside the process after its execution terminates; and the *inout* mode describes a modifiable parameter, which has an initial value that may be modified by a process locally and is visible after the process termination. For the behavior description B, this chapter highlights behaviors related to the modeling of communication, concurrency, and certain control structures. The complete description of behaviors is shown in Chapter 7 of [34].

2.2.3 Channel-Level Communication in LNT

The LNT language uses multiway gate rendezvous with data exchange to model communications between different processes. Two-way gate rendezvous can be used for modeling the channel-level communication. The behavior expression "G [$(O_0, \ldots, O_n)$] [**where** V]" specifies a potential rendezvous on gate G. Data exchange at a gate rendezvous is described as a list of offers $O_0, \ldots, O_n$. The optional condition expression V has to evaluate to **true** for the gate rendezvous to take place, and this expression uses values received by the offers $O_0, \ldots, O_n$. An offer is either output or input, defined as:

$$
\begin{aligned}
O \quad ::= \quad & [\ X \ \texttt{=>}\ ]\ [\ \texttt{!}\ ]V \\
| \quad & [\ X \ \texttt{=>}\ ]\ \texttt{?}P
\end{aligned}
$$

where an output offer, "[!]V", describes an emission of value expression V and an input offer, "?P", corresponds to the reception of a value matching pattern P. There are two effects of the pattern matching of a value V with a pattern P: it returns a **true** if V has the same structure as P, otherwise it returns a **false**; if V matches P, the variables used by P are initialized with the values extracted from V. Detailed grammar of patterns and their corresponding matching effects are provided in Chapter 6 of [34].

The semantics of gate rendezvous is that the communication is blocked by values in the offers on both sending and receiving ends waiting for the rendezvous, and the corresponding process executions are suspended until the rendezvous takes place. In general, LNT does not differentiate the sender and the receiver, and it is possible to have both sending and receiving offers on one gate.

In order to specify the connections among multiple processes, the parallel composition construct can be used. The grammar for the parallel composition has the following form:

par [G_0 **,** ... **,** G_n **in**]

[$G_{(0,0)}$ **,** ... **,** $G_{(0,n_0)}$ `->`] B_0

|| ... ||

[$G_{(m,0)}$ **,** ... **,** $G_{(m,n_m)}$ `->`] B_m

end par

Each gate specified by the optional *global synchronization gate set* "$\{G_0, ..., G_n\}$" requires that it appears in the gate declaration in every behavior that participates in the parallel composition. A global synchronization gate communication occurs if and only if all behaviors "$B_0, ..., B_m$" can make this communication simultaneously. Omission of the global synchronization gate set means no such gate exists in the parallel composition. Each behavior $B_i (\forall i \in [0, m])$ has an optional *synchronization interface* consisting of the set of gates $\{G_{(i,0)}, ..., G_{(i,n_0)}\}$. A gate communication in a behavior's synchronization interface can happen if and only if all behaviors specifying this gate in their respective communication interfaces can make this communication simultaneously. Omission of a synchronization interface of behavior B_i means that its synchronization is empty. The global synchronization gate set is a shorthand notation for gates that appear in every behavior's synchronization interface. Any behavior in the parallel composition can instantiate a process with the form:

Π [`[`*actual_gates*`]`] [`(`*actual_parameter*$_1$ **,** ... **,** *actual_parameter*$_n$`)`]

where process Π is instantiated with its gates and optional parameters substituted by *actual_gates* and a list of actual parameters.

A producer-consumer example is shown in Figure 2.2. The top-level LNT module,

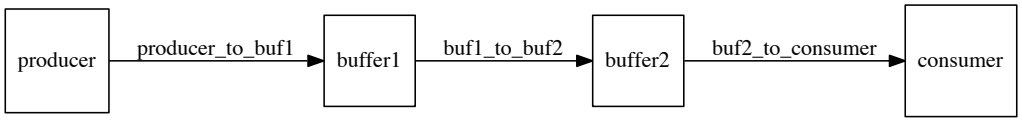


Figure 2.2: Block diagram for the producer-consumer example.

"producer_consumer", specified in Figure 2.3, implements this block diagram by instantiating one producer, two buffer, and one consumer processes, which are specified in Figure 2.4. It connects these instantiated processes through actual gates listed in their respective synchronization interfaces. Note that one of the modules in a model specification must be the principle module where the root process is included. The root process is a process with no value parameters and is named *main*. Each of the producer, the buffer, and the consumer processes has a *loop* construct that makes the process repeat infinitely without exiting. The producer process first declares an internal variable data, and then initializes it with a *nondeterministic assignment* that returns a value of type Nat ranging from 0 to 3. It then emits this value as an offer on the p_to_b gate. Connected to this gate is the input gate buf_in of the buffer process. Both gates are replaced by the actual gate producer_to_buf1 when their processes are instantiated in the producer_consumer module. Both processes have to simultaneously agree on a rendezvous on this gate. The input offer in a gate rendezvous uses pattern matching, which only admits offers that match its specified pattern. If the buffer process expected a Boolean input value instead, such as buf_in(?**any** Bool), the gate rendezvous could never take place due to matching failure. A successful pattern matching assigns values of the sender's offers to variables used by the receiver's pattern. In this example, a rendezvous on gate producer_to_buf1 assigns the value stored in the producer's local variable data to the first buffer's local variable data. On receiving the value from the producer, this buffer is ready to relay it to the second buffer by doing a gate rendezvous between its output gate buf_out and the second buffer's input gate buf_in. In a similar fashion, the value is finally passed to the consumer. Note that in the buffer process, a *sequential composition* of two rendezvous gates, i.e., "buf_in(?data); buf_out(data)", implies that buf_in(?data) is executed first, and if it terminates normally, then buf_out(data) is executed with the variable value updated by buf_in(?data). This ensures that the correct value is passed from the input to the output of a buffer.

Another feature of the parallel composition construct is that all behaviors "$B_0, ..., B_m$" placed in parallel run concurrently. For example, a gate rendezvous on producer_to_buff1 occurs between producer and the first buffer, and simultaneously the second buffer and consumer may communicate on gate buff2_to_consumer.

2.2.4 Nondeterministic Choice in LNT

The nondeterministic choice among a finite number of behaviors B_1, ..., B_n is expressed as **select** B_1 [] ... [] B_n **end select** in LNT. It first executes all choices of behaviors

```
module top(producer, consumer, buffer) is
process main [producer_to_buf1, buf1_to_buf2, buf2_to_consumer : any] is
   par
      producer_to_buf1 ->
         producer[producer_to_buf1]
   ||
      producer_to_buf1, buf1_to_buf2->
         buffer[producer_to_buf1, buf1_to_buf2]
   ||
      buf1_to_buf2, buf2_to_consumer ->
         buffer[buf1_to_buf2, buf2_to_consumer]
   ||
      buf2_to_consumer ->
         consumer[buf2_to_consumer]
   end par
end process
end module
```

Figure 2.3: Top-level LNT process for the producer-consumer example.

```
module producer is
process producer [p_to_b : any] is
   loop
      var data : Nat in
         data := any Nat where (data <= 3);
         p_to_b(data)
      end var
   end loop
end process
end module

module buffer is
process buffer [buf_in, buf_out : any] is
   loop
      var data : Nat in
         buf_in(?data);
         buf_out(data)
      end var
   end loop
end process
end module

module consumer is
process consumer [b_to_c: any] is
   loop
      b_to_c(?any Nat)
   end loop
end process
end module
```

Figure 2.4: Producer, buffer, and consumer LNT processes for the producer-consumer example.

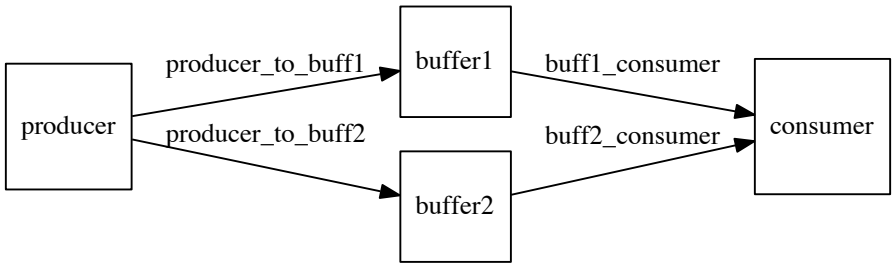


Figure 2.5: Block diagram for the modified producer-consumer example.

B_1, ..., B_n, and the first action (i.e., gate rendezvous, internal action, or termination) executed by one of the behaviors determines the result of the choice.

As an example, the producer-consumer example is modified to give both the producer and consumer a choice to send to or receive from two data buffers. The block diagram is shown in Figure 2.5. The LNT specification is presented in Figures 2.6 to 2.8. The producer process now repeatedly selects to communicate with one buffer: the data are sent only to a buffer whose input gate is ready for synchronization; if both input gates of the two buffers are ready, the choice is nondeterministic; if neither gate is ready, then the producer process blocks until one buffer's input gate becomes ready. Note that the choice is nondeterministic only when both buffer's inputs are simultaneously ready to communicate with the producer. It is, however, deterministic when only one buffer's input is ready.

If one wants to model a choice that is independent of the gate rendezvous availability, an internal action i can be inserted as the first action for each choice behavior. In the producer process, both internal actions are always ready to execute without needing to wait

```
module producer is
process producer [p_to_b1, p_to_b2 : any] is
   loop
      var data : Nat in
         data := any Nat where (data <= 3);
         select
            p_to_b1(data)
         []
            p_to_b2(data)
         end select
      end var
   end loop
end process
end module
```

Figure 2.6: Producer LNT process with nondeterministic choice.

```
module consumer is
process consumer [b1_to_c, b2_to_c: any] is
loop
  select
    b1_to_c(?any Nat)
  []
    b2_to_c(?any Nat)
  end select
end loop
end process
end module
```

Figure 2.7: Consumer LNT process with nondeterministic choice.

```
module top(producer, consumer, buffer) is
process main [producer_to_buf1, producer_to_buf2,
  buf1_to_consumer, buf2_to_consumer : any] is
   par
      producer_to_buf1, producer_to_buf2 ->
         producer[producer_to_buf1, producer_to_buf2]
   ||
      producer_to_buf1, buf1_to_consumer ->
         buffer[producer_to_buf1, buf1_to_consumer]
   ||
      producer_to_buf2, buf2_to_consumer ->
         buffer[producer_to_buf2, buf2_to_consumer]
   ||
      buf1_to_consumer, buf2_to_consumer ->
         consumer[buf1_to_consumer, buf2_to_consumer]
   end par
end process
end module
```

Figure 2.8: Top-level LNT process with nondeterministic choice for the modified producer-consumer example.

to synchronize with other gates. This is shown in Figure 2.9. Once an internal action is executed to resolve the choice, the producer only waits on communicating with one buffer, even though the other one might already be ready to synchronize.

2.2.5 Conditional Behaviors and Repetition in LNT

Conditional behaviors are usually used to model systems with conditional flow controls. The most common conditional behaviors in LNT are the *if* and *case* constructs. The *if* construct has the following grammar:

```
module producer is
process producer [p_to_b1, p_to_b2 : any] is
   loop
      var data : Nat in
         data := any Nat where (data <= 3);
         select
            i;
            p_to_b1(data)
         []
            i;
            p_to_b2(data)
         end select
      end var
   end loop
end process
end module
```

Figure 2.9: Producer LNT process with internal nondeterministic choice.

[**only**]**if** V_0 **then** B_0

[**elsif** V_1 **then** B_1

...

elsif V_n **then** B_n]

[**else** B_{n+1}]

end if

where each of the expressions $V_0, ..., V_n$ must evaluate to Boolean type. Another conditional behavior is the *case* construct. It is equipped with the pattern-matching feature like most functional languages. The *case* behavior has the following form:

case V **in**

[**var** $var\_declaration_0, ..., var\_declaration_n$ **in**]

$match\_clause_0$ -> B_0

| ...

| $match\_clause_m$ -> B_m

end case

where the first behavior whose matching clause evaluates to **true** is executed. A match clause has either a specific or generic pattern. The generic pattern is identified by the keyword **any**:

$$match\_clause \ ::= \ P_0 \ [\ \textbf{where}\ V_0\ ] \ |\ ...\ |\ P_n\ [\ \textbf{where}\ V_n\ ]$$
$$|\ \ \textbf{any}\ [\ \textbf{where}\ V\ ]$$

```
module producer is
process producer [p_to_b1, p_to_b2 : any] is
   var data : Nat in
      data := 3;
      loop
         case data in
            Succ (data) -> p_to_b1(data)
         |
            any ->
               p_to_b2(data);
               data := 3
         end case
      end loop
   end var
end process
end module
```

Figure 2.10: Producer LNT process with the case behavior.

An expression matches "P **where** V" if it first matches the pattern P, and the evaluation of V in the context of variables bound by the matching returns the Boolean value **true**.

Figure 2.10 shows a producer process with the case behavior. The process starts by assigning an initial value of 3 to variable data. It then repeatedly executes the *case* behavior to perform pattern matching on the value of data. The function *Succ* takes a natural number and returns its successor number. It is an example of a *constant pattern*. Pattern-matching of a value V with a constant pattern $F(P_0, \ldots, P_n)$ returns a Boolean value **true** if it is equal to the value, or **false** otherwise. To make it clear to explain, let us rename the argument of this function to $data'$ for the moment. For the value of $data$ to match this constant pattern, it has to be equal to the value returned by this function, i.e., $data == Succ(data')$. It is obvious that $data'$ has to be the predecessor of $data$ to satisfy this condition. The *case* pattern $Succ(data)$ then binds the value of variable $data$ to that of $data'$, effectively reassigning $data$ the value of its own predecessor. Since the behavior is a repeated loop, this value is reused for the next iteration of the case behavior. Matching of the "Succ(data)" pattern keeps occurring in each iteration, which decrements the value of "data" by 1 until it becomes 0. After its value reaches 0, the next iteration of the case behavior fails to match this pattern since no natural number's successor is 0. Matching of the generic pattern occurs instead and the value is reset to 3. The behavior of this producer is that it repeatedly does the following sequence of steps: it first sends values 2, 1, 0 on gate "p_to_b1", and then sends value 0 on gate "p_to_b2".

The LNT language supports several repetition behaviors. The previous examples have shown the *forever loop*, i.e. "**loop** B_0 **end loop**". This behavior is useful especially for modeling hardware processes, which usually execute their instructions forever. On the

```
module producer is
process producer [p_to_b1, p_to_b2 : any] is
   loop
      var data : Nat in
         data := 3;
         loop L in
            case data in
               Succ (data) where data > 1 -> p_to_b1(data)
            |
               Succ (data) where data == 1 -> p_to_b2(data)
            |
               any -> break L
            end case
         end loop -- L
      end var
   end loop
end process
end module
```

Figure 2.11: Producer LNT process with repetition behaviors.

other hand, a breakable loop "**loop** L **in** B_0 **end loop**" can exit on a loop break behavior "**break** L". Conditional loops include *while* and *for* loops, and their definitions and semantics are detailed in [34]. Figure 2.11 shows a breakable loop L nested inside a forever loop for a producer process. It exits when the value of "data" decrements to 0. This producer repeatedly sends value 2 on gate "p_to_b1", and then sends value 1 on gate "p_to_b2".

2.3 Behavioral Modeling and Verification in LEMA

The LPN *Embedded Mixed-signal Analyzer* (LEMA) tool has been developed for the formal modeling and verification of speed-independent asynchronous circuits [57], timed circuits [58–60], analog-and-mixed-signal circuits [61, 62], assembly language software [37, 63], and genetic circuits [64, 65]. This tool supports, among many others, models described in the channel-level VHDL [66]. Drawing inspirations from CSP, the channel model added the notion of the probe operations [67]. VHDL stands for *VHSIC (Very High Speed Integrated Circuits) Hardware Description Language*. It was originally developed by the U.S. Department of Defence as a means to document the behavior of integrated circuits in the mid-1980s. It has become one of the industry's standard languages for describing digital systems. It has built-in parallelism to assist concurrency modeling of hardware systems. Taking a high-level VHDL model description, the LEMA tool compiles it into LPNs [68, 69] that are used for model checking as shown in Figure 2.12. State reachability analysis is performed on the set of LPNs, with the option of applying on-the-fly state reduction techniques such as *automatic abstraction* [37], *partial order reduction*, and *compositional minimization* [57]. If a *deadlock*, or a violation of a *safety property* is found, a counterexample consisting of a state-transition

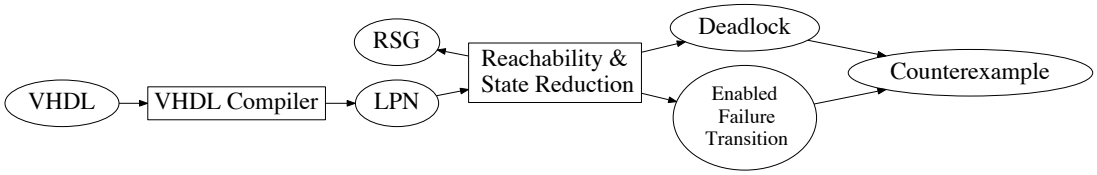


Figure 2.12: LEMA tool flow.

sequence that leads to the failure state is returned. Deadlock is reported when a state without any outgoing transitions is found. The safety property that can be checked during the reachability analysis is encoded by a *failure transition*, and it is the enabling of such a transition that causes a safety property violation. Alternatively, a *Reduced State-transition Graph* (RSG) is generated from the reachability analysis.

2.3.1 Channel-Level Communication in VHDL

The channel-level communication in VHDL describes point-to-point communication involving data exchange between two concurrently operating processes. A channel provides a means for two-way communication: it allows both participants to perform synchronization operations for passing data from one end to the other. Figures 2.13 to 2.16 show the VHDL description for the same producer-consumer LNT example in Figures 2.3 and 2.4, whose block diagram is shown in Figure 2.2. The first three lines of comments in Figure 2.13 indicate the name of the file.[2] The next line indicates that the standard IEEE library is used, and it is followed by a line defining the standard logic data type. The next two lines include the *nondeterminism* and the *channel* packages which are described in Appendix A of [66]. The nondeterminism package defines functions for producing random delays or selections for simulations. The channel package defines data types and operations on channels.

The remaining part of this VHDL file describes a design *entity* that consists of an *entity declaration* and an *architecture body*. The entity declaration is used to describe the interface of a design. In this example, it is simply empty because the entire system does not have any inputs or outputs. The architecture body describes the entity in one of the following ways: behavioral, structural, or a combination of both. It consists of a *declaration section* and a *concurrent statement section*. The declaration section can include definitions of *signals*, *constants*, etc., as well as *components* that declare other lower-level entities that can be

[2]In VHDL, comments start with "-" and extend to the end of the line.

```
-------------------------
-- top-level entity --
-------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;

entity top is
end top;
architecture structure of top is
  component producer
    port(p_to_b : inout channel);
  end component;
  component buf
    port(buf_in  : inout channel;
         buf_out : inout channel);
  end component;
  component consumer
    port(b_to_c : inout channel);
  end component;
  signal producer_to_buf1 : channel := init_channel;
  signal buf1_to_buf2 : channel := init_channel;
  signal buf2_to_consumer : channel := init_channel;
begin
  THE_PRODUCER : producer
    port map(p_to_b => producer_to_buf1);
  BUF1 : buf
    port map(buf_in  => producer_to_buf1,
             buf_out => buf1_to_buf2);
  BUF2 : buf
    port map(buf_in  => buf1_to_buf2,
             buf_out => buf2_to_consumer);
  THE_CONSUMER : consumer
    port map(b_to_c => buf2_to_consumer);
end structure;
```

Figure 2.13: Top-level VHDL entity for the producer-consume example.

instantiated. Like the LNT language, hierarchical modeling support in VHDL allows entities to be instantiated and composed by a top-level structural entity. Three components, namely "producer", "buf", and "consumer", are declared by this top-level entity "top". The entity declaration for the producer specifies its interface by declaring its port "p_to_b1". The component declaration is followed by signal declarations of the three channels that are shared between the producer and the first buffer, the first buffer and the second buffer, and the second buffer and the consumer, respectively, which correspond to the labels on the block diagram shown in Figure 2.2. The concurrent statement part makes instantiations of one producer, two buffers, and one consumer. An instantiation begins with the instance name followed by a component name. For example, the first buffer is named "*BUF1*" and is an instance of the *buf* component. The next part is the *port map*, which associates ports in each instantiated component with the signals declared in the top-level entity. The "buf_in" port of the "*BUF1*" instance is connected to "producer_to_buf1", which also connects the

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;

entity producer is
  port(p_to_b : inout channel := init_channel);
end producer;
architecture behavior of producer is
  signal data : std_logic_vector(1 downto 0) := "11";
begin
  producer : process
  begin
    data <= selection(4,2);
    wait for delay(5,10);
    send(p_to_b, data);
  end process producer;
end behavior;
```

Figure 2.14: Producer VHDL entity for the producer-consume example.

the "p_to_b" port of the producer instance named "*THE_PRODUCER*". The rest of the instantiations for other entities are similar.

The VHDL description for the producer is shown in Figure 2.14. The entity declaration section for the producer specifies its interface by declaring its port: *p_to_b*. A port declaration defines its data flow direction, i.e. *in* for an input, *out* for an output, and *inout* for either an input or an output. In this example, every channel is set to *inout* to allow signals to flow in both directions. Bidirectional flow is necessary because although data always flows in one direction, the handshake communication signals on each channel flow in both directions. The next part of the port declaration is the port types. The port for the "producer" entity has a *channel* type, and is initialized by the function call *init_channel*. Port initialization here is optional. Similar port declarations are specified in the entity declarations for the consumer and buffer entities, respectively. The architecture body begins with a signal definition, "data", that stores the value that is passed between communicating blocks. It is of type *std_logic_vector* which is an array of *std_logic* signals. The std_logic type has nine values, including the binary values 0 and 1, unknown value X, the uninitialized value U, the "don't care" value "-", the high impedance value Z, and symbols indicating weak strength signals (e.g., L for weak 0, H for weak 1, and W for weak unknown). The std_logic type is defined in the *std_logic_1164* package of the IEEE library. The value for each of the signals is encoded by a 2-bit-wide *std_logic_vector*. The value stored in "data" is sent out by the producer, and is initialized to "00".

The concurrent statement section, started with "**begin**" and ended with "**end** behavior", describes the behavior of each entity in this example with a *process statement*. A process

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;

entity buf is
  port(buf_in  : inout channel := init_channel;
       buf_out : inout channel := init_channel);
end buf;
architecture behavior of buf is
  signal data : std_logic_vector(1 downto 0);
begin
  buf : process
  begin
    receive(buf_in, data);
    send(buf_out, data);
  end process buf;
end behavior;
```

Figure 2.15: Buffer VHDL entity for the producer-consume example.

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;

entity consumer is
  port(b_to_c : inout channel := init_channel);
end consumer;
architecture behavior of consumer is
  signal data : std_logic_vector(1 downto 0);
begin
  consumer : process
  begin
    receive(b_to_c, data);
    --wait for delay(1, 1);
  end process consumer;
end behavior;
```

Figure 2.16: Consumer VHDL entity for the producer-consume example.

statement begins with an optional label followed by the keyword "**process**". Each process statement runs concurrently, but the statements within a process are executed sequentially. Also, sequential execution of these statements in a process loop forever. The buffer and consumer processes shown in Figures 2.15 and 2.16 have similar structures. Two procedure calls appearing in the behavior description of this example are *send* and *receive*. Each procedure takes two parameters: a channel to communicate on and a *std_logic* or *std_logic_vector* signal with a value to be transmitted. Both procedures are defined in the channel package.

The producer first randomly assigns a value to "data" by calling the *selection* function defined in the nondeterminism package. It takes two integer parameters: the first is the number of choices, and the second is the number of bits of the returned *std_logic_vector*. In this example, " selection (4,2)" returns a two-bit *std_logic_vector* with four possible choices.

Therefore, the value of "data" can be "00", "01", "10", or "11". After a value is assigned to "data", the producer calls the *delay* function to wait for a random period of time ranging from 5 to 10 time units. Defined in the nondeterminism package, this function takes two integer parameters as its lower and upper bound, and returns an integer number randomly drawn from the uniformly distributed range that is set by the given bounds. The producer initiates the communication with "BUF1" by calling the *send* procedure to transmit the value of "data" on channel "p_to_b". Execution of this procedure waits, i.e., blocks the sequential statements execution in the producer process, until the "BUF1" process is ready to receive the value on the same channel, at which time they synchronize to allow the producer to send the value to "BUF1". The "BUF1" process calls the *receive* procedure to complete the communication by accepting the value from the producer and storing it in "data". On successfully receiving this value, the "BUF1" process then communicates with "BUF2" by calling the *send* procedure to initiate transmission of the value of "data2" on channel "buf1_to_buf2". Following the similar behavior, data transmission from "BUF1" to "BUF2" and from "BUF2" to "THE_CONSUMER" is achieved through synchronization of the respective *send* and *receive* pairs. Note that since each process runs concurrently, communication between "THE_PRODUCER" and the "BUF1" can happen while simultaneously communication between the "BUF2" and "THE_CONSUMER" happens. For the purpose of simulation, it is necessary to have either a wait statement or a sensitivity list in every VHDL process so that no single process execution dominates forever, starving other processes. An implicit wait statement is included in both the send and receive procedures.

2.3.2 Probe and Parallel Communication

To model a behavior with nondeterministic choice of channel communications, it is necessary to introduce the *probe* function that is defined in the channel package. Taking a channel as a parameter, it returns either **true** if there is a pending communication on that channel, or **false** otherwise. This function allows an entity to peek a channel in order to determine if it needs to respond to a pending communication on that channel. With the *probe* function, one can model behaviors with nondeterministic choice.

The VHDL specifications in Figures 2.17 to 2.19 model the same LNT behavior presented in Figures 2.6 to 2.8. After assigning a random value to "data" and then waiting for a delay, the producer first checks for any pending communications on both of its channels "p_to_b1" and "p_to_b2". This is achieved with the procedure call to *await_any*, which waits until any of the channels have pending communications. If neither one has a pending communication, it halts the producer process until at least one pending communication is

```
entity producer is
  port(p_to_b1 : inout channel := init_channel;
       p_to_b2 : inout channel := init_channel);
end producer;
architecture behavior of producer is
  signal data : std_logic_vector(1 downto 0) := "00";
begin
  producer : process
  variable z : integer;
  begin
    data <= selection(4,2);
    wait for delay(5,10);
    await_any(p_to_b1, p_to_b2);
    if (probe(p_to_b1) and probe(p_to_b2)) then
      z := selection(2); -- returns either a 1 or a 2
      if (z = 1) then
        send(p_to_b1, data);
      else
        send(p_to_b2, data);
      end if;
    elsif (probe(p_to_b2)) then
      send(p_to_b2, data);
    else
      send(p_to_b1, data);
    end if;
    wait for delay(1, 1);
  end process producer;
end behavior;
```

Figure 2.17: Producer VHDL entity for the producer-consumer example with nondeterministic choice.

detected. This behavior is described by a *wait statement* that is comprised of a keyword *wait until* and a Boolean condition. Only when this formula evaluates to **true** can the producer process continue to execute the rest of its behavior. Once a pending communication on a channel is detected, the producer sends its data on that channel. This behavior is modeled by the *if* statement with conditions that are Boolean connections of the probing result on each channel. The *if* statement is a type of *selection* statement that is used to specify conditional flow controls. Other VHDL control structures are introduced in Section 2.3.3. When pending communications on both channels are detected, the producer randomly selects a channel with equal probability: the "selection (2)" function returns a value of either 1 or 2 with equal probability, based on which decision is made to communicate with a buffer. When a pending communication is detected only on one channel, the producer sends its data on that channel. Note that in the producer process, the two *if* statements cover all possible value combinations so that one of them is guaranteed to execute. Otherwise, an explicit wait statement needs to be added to avoid starving other processes during simulation.

```
entity consumer is
  port(b1_to_c : inout channel := init_channel;
       b2_to_c : inout channel := init_channel);
end consumer;
architecture behavior of consumer is
  signal data : std_logic_vector(1 downto 0);
begin
  consumer : process
    variable z : integer;
  begin
    await_any(b1_to_c, b2_to_c);
    if (probe(b1_to_c) and probe(b2_to_c)) then
       z := selection(2); -- returns either a 1 or a 2
       if (z = 1) then
         receive(b1_to_c, data);
       else
         receive(b2_to_c, data);
      end if;
    elsif (probe(b2_to_c)) then
      receive(b2_to_c, data);
    else
      receive(b1_to_c, data);
    end if;
    wait for delay(1, 1);
  end process consumer;
end behavior;
```

Figure 2.18: Consumer VHDL entity for the producer-consumer example with nondeterministic choice.

```
entity top is
end top;
architecture structure of top is
  component producer
    port(p_to_b1 : inout channel;
         p_to_b2 : inout channel);
  end component;
  component buff
    port(buff_in  : inout channel;
         buff_out : inout channel);
  end component;
  component consumer
    port(b1_to_c : inout channel;
         b2_to_c : inout channel);
  end component;
  signal producer_to_buff1 : channel := init_channel;
  signal producer_to_buff2 : channel := init_channel;
  signal buff1_to_consumer : channel := init_channel;
  signal buff2_to_consumer : channel := init_channel;
begin
  THE_PRODUCER : producer
    port map(p_to_b1 => producer_to_buff1,
             p_to_b2 => producer_to_buff2);
  BUFF1 : buff
    port map(buff_in  => producer_to_buff1,
             buff_out => buff1_to_consumer);
  BUFF2 : buff
    port map(buff_in  => producer_to_buff2,
             buff_out => buff2_to_consumer);
  THE_CONSUMER : consumer
    port map(b1_to_c => buff1_to_consumer,
             b2_to_c => buff2_to_consumer);
end structure;
```

Figure 2.19: Top-level VHDL entity for the producer-consumer example with nondeterministic choice.

```
entity producer is
  port(p_to_b1 : inout channel := init_channel;
       p_to_b2 : inout channel := init_channel);
end producer;

architecture behavior of producer is
  signal data : std_logic_vector(1 downto 0) := "00";
  signal selected_chan : std_logic_vector(0 downto 0);
begin
  producer : process
  variable z : integer;
  begin
    data <= selection(4,2);
    z := selection(2);
    wait for delay(5,10);
    if (z = 1) then
      send(p_to_b1, data);
    else
      send(p_to_b2, data);
    end if;
    --wait for delay(1, 1);
  end process producer;
end behavior;
```

Figure 2.20: Producer VHDL entity for the producer-consumer example with internal nondeterministic choice.

The behavior of the consumer, shown in Figure 2.18, is symmetrical to that of the producer: it waits until a buffer requests a communication; it has a choice of receiving from either of the two buffers, depending on which one is available to communicate, and a random choice is made equally between them when both are ready. Note that the uniform random distribution is deployed for the realization of the nondeterministic choice. It is possible to assign other random distributions to model nondeterminism. The buffer's behavior remains the same as the previous example.

To model a nondeterministic choice independent of a channel's availability to communicate, the producer can be modified such that it selects a channel to communicate first, after which it waits for a pending communication on the selected channel before sending the data. The behavioral description for the producer is shown in Figure 2.20. The integer signal "z" stores the two possible outcomes of the random choice: 1 or 2. If the value is 1, communication is expected on channel "p_to_b1", otherwise channel "p_to_b2" is expected to deliver the data.

2.3.3 VHDL Control Structures

For system models with conditional flow controls, VHDL provides *selection* and *repetition* statements. Two *selection* statements of interest are the *if* statement and *case* statement. The *repetition* statements in VHDL include the *infinite loop*, the *while-loop*, and the *for-loop*.

```
entity producer is
  port(p_to_b1 : inout channel := init_channel;
       p_to_b2 : inout channel := init_channel);
end producer;

architecture behavior of producer is
  signal data : std_logic_vector(1 downto 0) := "11";
begin
  producer : process
  begin
    case data is
      when "11" | "10" | "01" =>
        data <= std_logic_vector(unsigned(data) - 1);
        wait for delay(1,1);
        send(p_to_b1, data);
      when others =>
        send(p_to_b2, data);
        data <= "11";
        wait for delay(1,1);
    end case;
    wait for delay(1, 1);
  end process producer;
end behavior;
```

Figure 2.21: Producer VHDL entity with a case statement.

The *if* statement was introduced with examples in Section 2.3.2. This section describes the following three statements: *case*, *infinite loop*, and *while-loop*, through several examples.

The *case* statement in VHDL checks the value of a single expression, based on which it executes the one statement whose choice matches that value. The modified producer example presented in Figure 2.21 is a reproduction of the LNT code in Figure 2.10. It initially assigns "data" the two-bit binary-encoded value 3, and then repeatedly executes the following steps: it sends values 2, 1, and 0 on channel "p_to_b1", then it sends value 0 on channel "p_to_b2", after which "data" is reset to 3. Comparing to LNT, the VHDL *case* statement lacks the pattern-matching feature, and does not allow a nonstatic expression (e.g., a function) in any of its choice clauses. The choice can only be either a static expression (e.g., 5, 4|6|8) or a range expression (e.g., 4 to 9). Therefore, the value decrement of the data needs to be explicitly expressed in "data <= std_logic_vector(unsigned(data) − 1)".

Since VHDL was designed to document the behavior of integrated circuits that mostly repeat forever, the language inherently implies an infinite loop for every process. It is also possible to specify a breakable infinite loop internal to a process. Figure 2.22 shows a modified producer that uses a breakable infinite loop. Exhibiting the same behavior as the LNT model in Figure 2.11, it repeatedly sends two-bit binary value 10 on channel "p_to_b1" and 01 on channel "p_to_b2". For other data values, a loop break "**exit** L" occurs. As a result, the consumer receives the alternation of both values. One can create an equivalent model with a *while-loop* statement together with a signal to exit the loop. An example

```
entity producer is
  port(p_to_b1 : inout channel := init_channel;
       p_to_b2 : inout channel := init_channel);
end producer;

architecture behavior of producer is
  signal data : std_logic_vector(1 downto 0);
begin
  producer : process
  begin
    data <= "11";
    wait for delay(1,1);
    L : loop
      case data is
        when "11" =>
          data <= std_logic_vector(unsigned(data) - 1);
          wait for delay(1,1);
          send(p_to_b1, data);
        when "10" =>
          data <= std_logic_vector(unsigned(data) - 1);
          wait for delay(1,1);
          send(p_to_b2, data);
        when others => exit L;
      end case;
    end loop;
    wait for delay(1,1);
  end process producer;
end behavior;
```

Figure 2.22: Producer VHDL entity with a breakable infinite loop statement.

VHDL entity is presented in Figure 2.23. The signal "breakloop" is used to determine whether the execution remains in the *while-loop* if its condition evaluates to **false**, or exists the loop if the condition evaluates to **true**.

2.3.4 LPN Syntax and Semantics

The high-level VHDL model description can be automatically compiled to a LPN formal representation by the LEMA tool. The conversion is necessary as the model-checking engines shown in Figure 2.12 are based on this formalism. This section describes the LPN model that is used to model the systems to be verified.[3]

A LPN is a *1-safe Petri net* in which the transitions have been labeled with expressions on auxiliary variables to represent when transitions can occur and how they update the system state. A LPN is a tuple $\langle P, T, F, M_0, B, X, S_0, Y_0, L, T_f, T_p, T_d \rangle$ where:

- P is a finite set of places;

[3]The version of the LPN model used in this chapter is a limited form of the one described in [37, 38]. In particular, this version does not include continuous variables and includes only a limited form of the delay assignments. This form is, however, sufficient to describe the asynchronous and concurrent systems described in this thesis.

```
entity producer_while is
  port(p_to_b1 : inout channel := init_channel;
       p_to_b2 : inout channel := init_channel);
end producer_while;

architecture behavior of producer_while is
  signal data : std_logic_vector(1 downto 0) := "11";
  signal breakloop : std_logic;
begin
  producer_while : process
  begin
    data <= "11";
    breakloop <= '0';
    wait for delay(1,1);
    L : while (breakloop = '0') loop
      case data is
        when "11" =>
          data <= std_logic_vector(unsigned(data) - 1);
          wait for delay(1,1);
          send(p_to_b1, data);
        when "10" =>
          data <= std_logic_vector(unsigned(data) - 1);
          wait for delay(1,1);
          send(p_to_b2, data);
        when others =>
          breakloop <= '1';
          wait for delay(1,1);
      end case;
    end loop L;
    wait for delay(1,1);
  end process producer_while;
end behavior;
```

Figure 2.23: Producer VHDL entity with a while-loop statement.

- T is a finite set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- $M_0 \subseteq P$ is the set of initially marked places;
- B is a finite set of Boolean variables;
- X is a finite set of integer variables;
- $S_0 : B \to \{0, 1\}$ is the initial value for each Boolean variable;
- $Y_0 : X \to \mathbb{Z}$ is the initial value for each integer variable;
- L is a tuple of labels defined below;
- $T_f \subseteq T$ is a finite set of *failure* transitions;
- $T_p \subseteq T$ is a finite set of *persistent* transitions;
- $T_d \subseteq (T - T_p)$ is a finite set of *disabling failure* transitions.

The first four elements define an ordinary Petri net. The places, P, represent the states, the transitions, T, represent the state transitions, and the flow relation, F, expresses the connectivity between the places and transitions. The initially marked places, M_0, are the initial states for each process. The next five elements define the auxiliary Boolean, B, and integer, X, variables, their initial values, S_0 and X_0, respectively, and the transition labels, L, that use these variables. Finally, the last two elements are used to indicate special transition types which are described in more detail below.

Before the labels are defined, the grammar used by the labels must first be presented. The grammar of a LPN can be categorized into the numerical portion and the Boolean portion. The numerical portion is defined as follows:

$$\begin{aligned} \chi \quad ::= \quad & c \mid x \mid (\chi) \mid -\chi \mid \chi + \chi \mid \chi - \chi \mid \chi * \chi \mid \chi / \chi \mid \chi^{\chi} \mid \\ & \mathrm{NOT}(\chi) \mid \mathrm{OR}(\chi, \chi) \mid \mathrm{AND}(\chi, \chi) \mid \mathrm{XOR}(\chi, \chi) \mid \mathrm{INT}(\phi) \end{aligned}$$

where c is a rational constant from $\mathbb{Q}$, x is an integer variable, and the arithmetic operators are defined as usual. The functions NOT, OR, AND, and XOR are bit-wise logical operations, and they are only applicable to integers and assume a 2's complement format with arbitrary precision. The function INT converts a Boolean **true** value to an integer 1 and **false** value to an integer 0. The set $\mathcal{P}_{\chi}$ is defined to be all formulas that can be constructed from the χ grammar. The Boolean portion of the grammar is defined as follows:

$$\begin{aligned} \phi \quad ::= \quad & \mathbf{true} \;\Big|\; \mathbf{false} \;\Big|\; b \;\Big|\; \mathrm{BIT}(\chi, \chi) \;\Big|\; \sim \phi \;\Big|\; \phi \;\&\; \phi \;\Big|\; \phi \mid \phi \;\Big| \\ & \chi = \chi \;\Big|\; \chi \geq \chi \;\Big|\; \chi > \chi \;\Big|\; \chi \leq \chi \;\Big|\; \chi < \chi \end{aligned}$$

where b is a Boolean variable, $\mathrm{BIT}(\alpha 1, \alpha 2)$ extracts bit $\alpha 2$ from $\alpha 1$, and the logical and relational operators are defined as usual. The set $\mathcal{P}_{\phi}$ is defined to be all formulas that can be constructed from the ϕ grammar.

The grammar used by the delay expression is defined as follows:

$$\psi \quad ::= \quad 0 \mid \mathrm{UNBOUNDED}$$

where 0 is an integer value 0, and the function UNBOUNDED indicates that the delay is unbounded and can be any value between 0 and $+\infty$. The set $\mathcal{P}_{\psi}$ is defined to be all formulas that can be constructed from the ψ grammar.

Each LPN transition is labeled with an enabling condition and a set of assignments which can now be defined as follows, $L = \langle \mathrm{En}, \mathrm{DA}, \mathrm{BA}, \mathrm{XA} \rangle$:

- $\mathrm{En} : T \rightarrow \mathcal{P}_{\phi}$ labels each transition $t \in T$ with an enabling condition.

- $DA : T \rightarrow \mathcal{P}_\psi$ labels each transition $t \in T$ with a delay expression.
- $\mathrm{BA} : T \times B \rightarrow \mathcal{P}_\phi$ labels each transition $t \in T$ and Boolean variable $b \in B$ with the Boolean assignment made to b when t fires.
- $\mathrm{XA} : T \times X \rightarrow \mathcal{P}_\chi$ labels each transition $t \in T$ and integer variable $x \in X$ with the integer variable assignment made to x when t fires.

Note that many assignments on a transition may be *vacuous* in that they do not change a variable's value (i.e., $\mathrm{BA}(t, b) = b$ or $\mathrm{XA}(t, x) = x$).

A strongly connected set of places and transitions in a LPN is referred to as a *process*. A system model can have a set of concurrent sequential *processes* that communicate through the variables in the labels. Indeed, in all of the models used in this thesis, each process includes only a single marked place in any state. Therefore, if preferred, the reader can consider the model as essentially a form of communicating *finite state machines*.

This section gives a brief description of the semantics focusing on the LPN model presented in the previous section. A more formal and complete LPN semantics description can be found in [37, 38]. A state of a LPN is a tuple $\langle M, E, S, Y \rangle$ where M is the current marking, E is the set of enabled transitions at this state, S is the current value of the Boolean variables, and Y is the current value of the integer variables. For any transition $t \in T$, its preset is denoted by $\bullet t = \{p \mid (p, t) \in F\}$ and postset is denoted by $t\bullet = \{p \mid (p, t) \in F\}$. Presets and postsets for places are defined similarly. The evaluation function, eval, is defined to take an expression and the values of the variables and return the evaluation of the expression. A transition is enabled in a state if it satisfies all of the following conditions:

1. All of the places in its preset are marked (i.e., $\bullet t \subseteq M$).
2. One of the following conditions is satisfied:
   (a) Its enabling condition function $\mathrm{En}(t)$, evaluates to **true** (i.e., $\mathrm{eval}(\mathrm{En}(t), S, Y) =$ **true**).
   (b) It is a persistent transition and has not been fired since it was enabled in a previous state.

Once a persistent transition is enabled, it cannot become disabled by its enabling condition becoming **false**. The only way to disable a persistent transition after it becomes enabled is to either remove the marking from its preset place or fire it.

Any enabled transition t can be selected to fire leading to a new state $\langle M', E', S', Y' \rangle$ from the current state $\langle M, E, S, Y \rangle$. The new state is defined as follows:

$$
\begin{aligned}
M' &:= (M - \bullet t) \cup t \bullet \\
S'(b) &:= \mathrm{BA}(t, b) \;\; \forall b \in B \\
Y'(x) &:= \mathrm{XA}(t, x) \;\; \forall x \in X \\
E' &:= \{t' \mid \bullet t' \subseteq M' \wedge \big((\mathrm{En}(t'), S', Y') = \mathbf{true}\big) \vee \big(t' \in T_p \Rightarrow (t' \in E \wedge t' \neq t)\big)\}
\end{aligned}
$$

Firing of a transition also depends on its type of delay expression: an *immediate* transition has a constant 0 delay and it gets fired immediately once enabled; an *unbounded* transition has a nonzero positive delay and can fire once enabled, but only after firings of all simultaneously enabled immediate transitions.

The aforementioned three transition types indicate differences in interpretation when a transition is enabled or disabled, allowing us to express correct behaviors and failures. The first type of failure occurs when a transition from the failure set, T_f, fires. Actually, it is sufficient to indicate a failure once it becomes enabled, since at that point, there certainly exists a failure as it can be chosen as the next transition to fire. Failure transitions are useful for describing a variety of different types of safety properties. After a transition fires and changes the state, the enabling condition of some enabled transitions may become disabled. The default behavior in this case is that these transitions are no longer considered to be enabled and cannot fire until their enabling condition becomes **true** again. If this transition is in the disabling failure set, T_d, this disabling is considered a failure. Disabling failure transitions are useful for describing circuit *hazards* (i.e., glitches on gates). In some cases, it is desirable to disallow the disabling of a transition in a model, and these transitions are members of the persistent transition set, T_p.

A simple producer-consumer model is shown in Figure 2.24. The model has only one communication channel `p_to_c` that transmits a value 3 from the producer to the consumer. The generated LPN shown on the right of this figure includes two processes. The process on the left models the producer while the process on the right models the consumer. This LPN includes places which are labeled $ip0$, $ip1$, $ip2$, $ip3$, $ip4$, and $ip5$. The places are marked to indicate the current state. Places $ip4$ and $ip5$ are initially marked. Transitions in this figure are labeled $p\_to\_c\_sendP1$, $p\_to\_c\_sendM1$, d_0P1, $p\_to\_c\_rcvP1$, $p\_to\_c\_rcvM1$, and d_1P1. The LPN also includes Boolean variables $p\_to\_c\_send$ and $p\_to\_c\_rcv$ which are both initially **false**, and integer variables $p\_to\_c\_data$, $d0$, and $d1$ that have initial values of 0, 3, and 0, respectively. Each transition is labeled

```
entity producer_consumer is
end producer_consumer;

architecture behavior of producer_consumer is
  signal p_to_c:channel:=init_channel;
  signal d0:std_logic_vector(1 downto 0):="11";
  signal d1:std_logic_vector(1 downto 0);
begin
  producer:process
  begin
    send(p_to_c,d0);
  end process producer;
  consumer:process
  begin
    receive(p_to_c,d1);
  end process consumer;
end behavior;
```

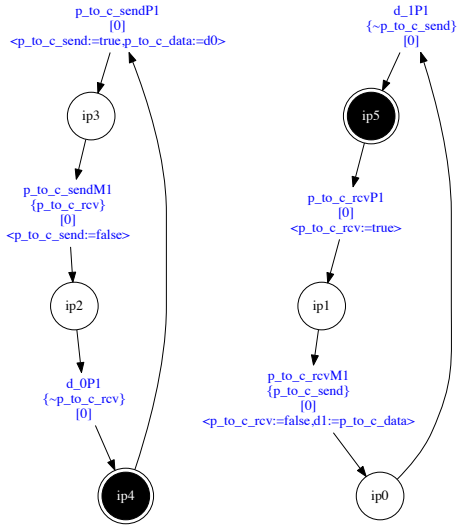
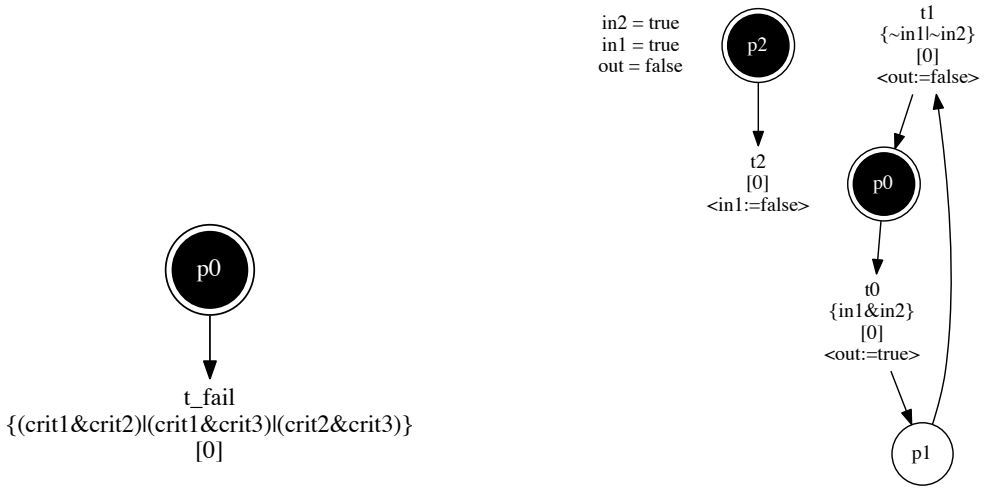


Figure 2.24: LPN for a simple producer-consumer model.

with an enabling condition, a delay expression, and a set of assignments. For example, transition $p\_to\_c\_rcvM1$ has enabling condition $\{p\_to\_c\_send\}$, delay expression [0] and assignments $\langle p\_to\_c\_rcv := \textbf{false}, d1 := p\_to\_c\_data\rangle$. Intuitively, these labels indicate that transition $p\_to\_c\_rcvM1$ is only enabled when $p\_to\_c\_send$ evaluates to **true**, and it fires immediately after it becomes enabled, and when it fires, it performs the specified assignments. The transition $p\_to\_c\_rcvM1$, though, is not enabled in the initial state because place $ip1$ is not initially marked. In the initial state, transitions $p\_to\_c\_sendP1$ and $p\_to\_c\_rcvP1$ are the only enabled transitions because they have **true** enabling conditions (default when not shown) and their input places (i.e., $ip4$ and $ip5$) are marked. Assume that $p\_to\_c\_sendP1$ fires first setting $p\_to\_c\_send$ to **true** and the integer variable $p\_to\_c\_data$ is assigned the value stored in $d0$, which is 3. The marking is also transferred from $ip4$ to $ip3$. In this new state, only transition $p\_t\_c\_rcvP1$ is enabled, since $p\_to\_c\_rcv$ is **false**. After firing $p\_to\_c\_rcvP1$, the value of $p\_to\_c\_rcv$ becomes **true** and $ip1$ is marked. In this state, both $p\_to\_c\_sendM1$ and $p\_to\_c\_rcvM1$ are enabled. Note that all transitions are persistent. Otherwise, the firing of one transition would disable the other, creating a race condition and leading to incorrect behavior of the model.

Figure 2.25a shows a LPN with a failure transition. Its enabling condition specifies that if any pair of the Boolean variables *crit1*, *crit2*, and *crit3* is **true**, the failure transition is enabled. This LPN can be used to check for violations of mutual exclusion when three processes compete for entering the critical section. In Figure 2.25b, the LPN for a two-input



(a) Failure transition.

(b) Two-input AND gate.

Figure 2.25: LPN examples with failure and disabling failure transitions.

AND gate has two disabling failure transitions, $t0$ and $t1$. Assume the two input signals are both **true** in the state shown in Figure 2.25b. Both transition $t0$ and $t2$ are enabled. It is possible that an input signal *in1* glitches if $t2$ fires before $t0$. The result is that the gate has a hazard which is typically considered a failure in an asynchronous circuit.

2.4 Conclusion and Discussion

This chapter presents detailed examples in both LNT and channel-level VHDL for the modeling of concurrent systems with emphasis on the channel-level communication, non-deterministic choice, parallel composition, as well as various data flow constructs. The formal definition and semantics of a subset of LPNs are described, followed by a channel-level communication example compiled from the VHDL description and examples indicating safety properties using different notions of failure transitions.

For the channel-level communication without probe, the model in VHDL corresponds to the two-way gate rendezvous in LNT. The offers on a LNT gate can take on all data types specified by its syntax while the data type on a VHDL channel is limited to either *std_logic* or *std_logic_vector*. It is, however, possible to convert other types in VHDL to either of these types with additional steps. The channel definition in VHDL uses CSP-like notation for events, which clearly distinguishes the "send" from the "receive" action. The LNT gate does not have this distinction which makes it capable to send and receive

multiple offers in one rendezvous. Gate rendezvous in LNT performs pattern matching that implements exchange of values in its offers. At the LPN level, the simple "send-and-receive" communication on a channel is represented with six transitions with two additional handshake signals (Figure 2.24), compared to one gate-synchronization action in LNT.

Another difference of channel communication is the probe operation in VHDL. LNT lacks direct support for this operation, although a complex implementation of the general probe operation is possible [70]. When multiple gates are used as choice behaviors in its nondeterministic construct, the choice is fair. In other words, no priority is inferred when a choice needs to be resolved between two gates that are simultaneously ready for communication. Implementing a priority choice is difficult as it requires probing the possibility of a gate rendezvous.

Pattern matching built in the *case* construct in LNT allows clause matching and possible variable assignments to be combined in one step, compared to two separate steps in VHDL. Despite these differences in modeling constructs and semantics, this dissertation shows that both can be used successfully to represent the behavior of a fault-tolerant NoC router.

CHAPTER 3

NoC ARCHITECTURE AND ROUTING ALGORITHM

This chapter introduces the motivating example for this research work: a NoC architecture that supports the link-fault-tolerant routing algorithm [33] extended to a multiflit wormhole routing setting. Various fault-tolerant routing design methodologies are surveyed first. This chapter then describes in detail the Glass and Ni routing algorithm [32] on a two-dimensional (2D)-mesh topology, followed by an analysis of limitations of its fault tolerance ability. In particular, cyclic deadlock exists on a link fault, and it fails to handle one-away link faults. The link-fault-tolerant algorithm [33] is then introduced to fix these issues, but it pays the price of packet drop to break cycles. Simulation results indicate that this algorithm provides significant improvements in network reliability with minimal cost. Lastly, an extended architecture for link-fault routing is suggested to allow simultaneous processing of multiple multiflit packets.

3.1 NoC and Fault-tolerant Routing

With the advances in process technology, multicore architecture has been replacing single-core architecture due to performance improvement. The NoCcommunication paradigm has been widely investigated for future *Multiprocessor Systems-on-Chips* (MPSoCs) and *Chip Multiprocessors* (CMPs) [71, 72]. Early works such as [73] adapted asynchronous designs to implement on-chip networks. Recent work on asynchronous NoCs [74–82] has demonstrated many advantages over the corresponding synchronous designs, such as elimination of complex clock distribution, ability to easily interface with multiple clock domains, and lower power consumption.

Communication in NoCs operates as exchanges of data packets between the communicating nodes. A packet follows a certain path to reach its destination through the network, and its path is determined by the routing algorithm. Routing algorithms, therefore, can significantly impact the communication performance of the entire NoC. Different routing algorithms exhibit different degrees of adaptivity. The nonadaptive routing algorithm chooses

one of the shortest paths to route a packet from the sender to the receiver. For example, XY-routing routes a packet along the x-direction first, then along the y-direction to its destination. Routing algorithms that allow some but not all packets to use any of the shortest paths are identified as partially adaptive. One example of a partially adaptive routing algorithm is the *turn model* routing algorithm introduced by Glass and Ni in [83]. This algorithm eliminates the formation of deadlocks by preventing particular turns in the network. Lastly, the fully adaptive routing algorithms route packets on any shortest path. Boura et al. [84] and Chien et al. [85] provide fully adaptive routing with the help of virtual channels.

Another important feature of a NoC routing algorithm is its tolerance to network faults. It requires such an algorithm to provide built-in redundancy in communication that can be used for hardware reconfiguration, so that an alternative link can be chosen to route around a dead node to maintain the normal operation of the system. This is accomplished by dynamically redirecting packets to avoid failures in the network. To tolerate permanent faults, a reconfigurable routing table, e.g., [86, 87], is deployed to precompute and store routes to avoid faulty links. This approach, however, cannot tolerate transient faults. Virtual channels have been extensively used to provide fault tolerance in various routing algorithms (e.g., [88–90]). Duato analyzes in [91] the effective redundancy for adaptive fault-tolerant routing algorithms, which requires at least four virtual channels per physical channel. Note that virtual channels are not free. The use of virtual channels introduces extra area and energy cost. Also, in the case of a single faulty physical link, all virtual links belonging to that faulty physical link become faulty as well. Nordbotten et al. [92] use intermediate nodes as a backup mechanism to route packets around network failures. This solution, however, requires extra virtual channels to handle deadlocks, and has to pause all running processes to identify intermediate nodes when a fault is encountered. Ideas of binding faulty links and nodes into faulty polygons, chains, and rings have been presented in [88, 90, 93]. Packets are routed around these faulty regions to achieve fault tolerance. However, a major problem of these approaches is that they create heavy traffic load for the nodes having to route packets around the faulty areas. Ideally, network traffic should be distributed in a balanced way to avoid possible congestion in certain areas. To obtain balanced traffic distribution, dedicated wires are employed by a node to provide traffic information of its neighboring nodes (e.g., [94, 95]). The node then makes routing decisions based on the obtained local routing information to avoid congestion. They are, however, only limited to local traffic information. The adaptive routing algorithm in [96] requires a node that is

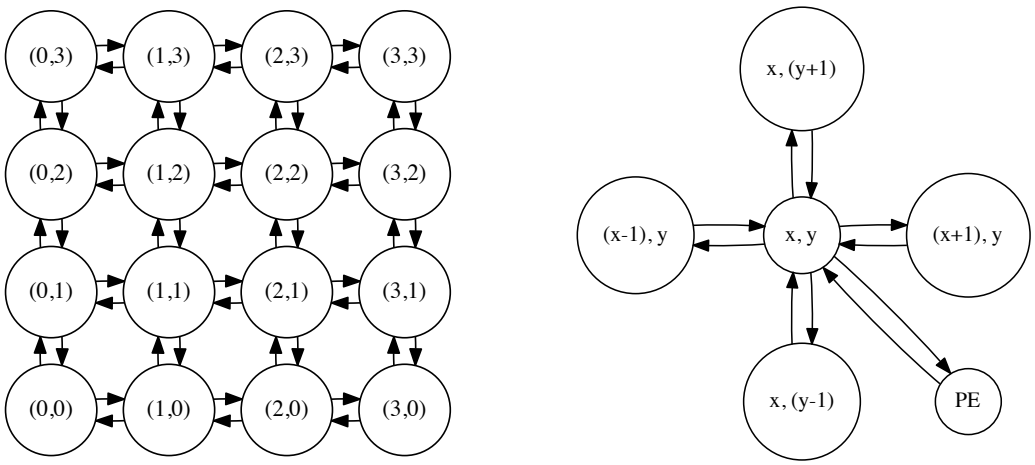
nearly overloaded to send stress values to its neighbor. The drawback is that the neighboring nodes may already send packets to the overloaded node before they receive the stress values. A different mechanism for exchanging routing information between nodes is proposed in [97]. Each node receives and updates the knowledge of all possible paths for a packet, which is used for uniformly distributing the traffic across the network.

Wu presented a fault-tolerant algorithm [98] extending Chiu's odd-even turn model [99]. Without the need of using virtual channels, the algorithm combines multiple faults to faulty-blocks and route packets around the faulty-blocks. In [32], Glass and Ni proposed an extension of the negative-first routing algorithm [83] that tolerates up to $(n-1)$ faults for n-dimensional meshes. This algorithm is partially adaptive and achieves deadlock freedom without any virtual channels. The algorithm always attempts to choose a route that allows a packet to have multiple choices in selecting its next routing node along the path to its destination. On a 2D mesh, if the first node cannot be reached due to a failure, the packet may use the second choice as the alternative path. Therefore, a single failed node can always be tolerated.

3.2 The Glass/Ni Routing Algorithm

Figure 3.1 shows an on-chip mesh network with 16 connected nodes and the structure of a routing node. Each node shown in Figure 3.1a is labeled by its position (x, y) and connects to each of its neighbors with a pair of input/output ports. Nodes have only three or two neighbors if they are on the edges or corners of the mesh, respectively. Figure 3.1b illustrates the structure of each routing node. It consists of a *processing element* (PE) and a router. The PE injects packets into the network and absorbs packets that are destined for its node (x, y). The router receives a packet, computes its next forwarding direction, and then sends the packet in that direction.

The Glass/Ni algorithm [32] implements the negative-first routing, and achieves deadlock freedom by disallowing certain turns in the network. In this turn model, the west and south boundaries of the network are defined as negative edges, and the north and east boundaries are defined as positive edges. Packets routed toward these two kinds of edges are called the negative and positive phases of routing, respectively. To route a packet, the algorithm requires the packet to proceed in negative directions first, until it reaches farther west and south than the destination; then the packet is allowed to move in the positive directions. The general rule is that once the packet is moving in the positive directions, it is forbidden to move in the negative directions. The only exception where positive-to-negative routing



(a) Network topology. (b) Structure of an individual node.

Figure 3.1: A 4-by-4 2D mesh network with 16 nodes.

(referred to as "illegal turn" in the following text) happens is when a faulty node on a negative edge blocks the path along the edge. The routing algorithm, however, strictly specifies all allowed illegal moves.

The Glass/Ni routing algorithm is proven to always tolerate one node fault in a two-dimensional mesh network [32]. This robustness is achieved by ensuring that a packet always has at least two choices of its next move at each node along its path. When on its negative phase, a packet can either stay in this phase or switch to the positive phase. If a packet is already on its positive phase, it still has two choices of going north or east. There is, though, one special case that needs to be considered, which is when a packet traverses a negative edge (i.e., the south edge or the west edge) of the network and encounters a faulty node. In this case, a packet only has one choice to route. For example, Figure 3.2 shows a portion of a mesh network where nodes (0,0), (1,0), and (2,0) are on the south edge of the network. Let us denote the forward link from node (x_1, y_1) to node (x_2, y_2) as $(x_1, y_1) \rightarrow (x_2, y_2)$. Suppose that a packet located at node (0,0) is destined for node (2,0). If no fault is encountered, the packet would follow the route $(0,0) \rightarrow (1,0) \rightarrow (2,0)$. If node (1,0) has a fault, the packet instead must route around the faulty node using the route $(0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (2,0)$. This route, though, includes the illegal turn from (1,1) to (2,0). For this special case, this illegal turn is allowed. However, it has no potential of introducing deadlock because no cycle can be formed in this case as the faulty node, (1,0)

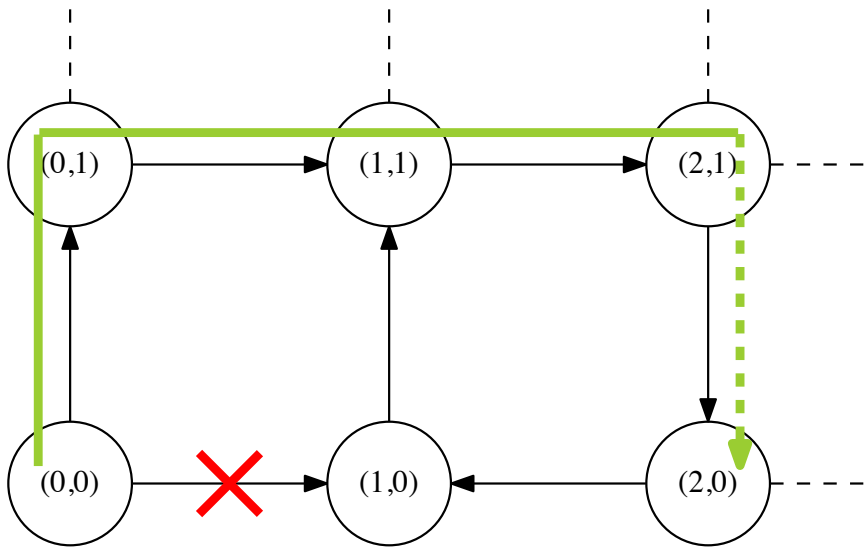


Figure 3.2: A special case for negative edges in the Glass/Ni algorithm.

in this example, breaks the potential cycle. To summarize the Glass/Ni routing algorithm, the computation of the direction obeys the following rules:

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

2. Route the packet east and north to the destination, avoiding routing the packet as far east or north as the destination for as long as possible.

The special case for the Glass/Ni algorithm just described makes the assumption of a node fault model. The node fault model treats a node's input link as part of the node, and once any part of a node is detected as faulty, all other parts of the node are considered faulty. This means that when a node's input link becomes faulty, the entire node stops operating immediately and becomes permanently unusable. This simplification overlooks the fact that a node can still function when its links fail. In reality, the occurrence of a single link fault is extremely rare, and having four faulty links on one node is almost unrealistic. It is, therefore, necessary to model link faults and node faults separately. Since the Glass/Ni routing algorithm only admits a node fault model, it is not directly applicable to the link-fault model, as not only does the algorithm not guarantee one-fault tolerance,

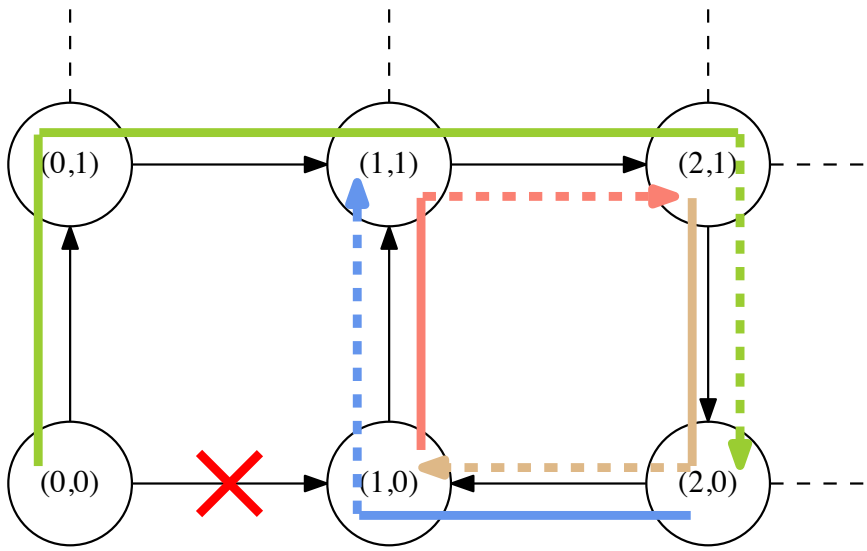


Figure 3.3: Deadlock caused by a link fault.

but also it potentially causes deadlock in the network. For example, Figure 3.3 illustrates a packet encountering a faulty link between nodes $(0,0)$ and $(1,0)$. As described earlier, the Glass/Ni algorithm detours packets from $(0,0)$ to $(2,0)$ using the route $(0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (2,0)$. Since it is the link $(0,0) \rightarrow (1,0)$ that is considered as faulty rather than node (1,0), cycles, such as $(1,0) \rightarrow (1,1)$, $(1,1) \rightarrow (2,1)$, $(2,1) \rightarrow (2,0)$, $(2,0) \rightarrow (1,0)$, can still be formed. Suppose each of these four links has a one-place buffer, and link $(1,0) \rightarrow (1,1)$, $(1,1) \rightarrow (2,1)$, $(2,1) \rightarrow (2,0)$, and $(2,0) \rightarrow (1,0)$ are obtained by four packets, which are respectively going to node $(2,1)$, $(2,0)$, $(1,0)$, and $(1,1)$. None of the four packets can make progress, as each packet is blocked by a packet in its outgoing channel. The result is a deadlock.

There is another special situation that cannot be handled by the Glass/Ni routing algorithm when assuming link faults. When a packet encounters a fault that is one hop away from its destination node, the Glass/Ni algorithm must drop the packet since it assumes the fault happens on the destination node. However, in the link-fault model, there are still available routes to the destination node. In [80], Yoneda and Imai address this problem by introducing a mechanism to forward the link-fault location to a neighboring routing node where selection of a faulty route is avoided. In the example shown in Figure 3.4, a packet is at node $(1,1)$, and its final destination node is $(2,2)$. Before sending out the packet,

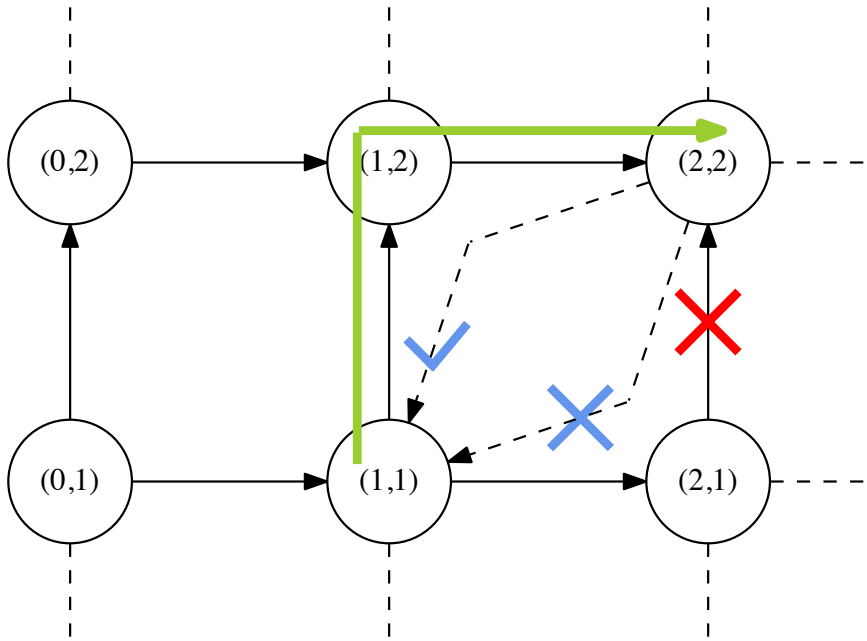


Figure 3.4: A fault lookahead mechanism.

node $(1,1)$ checks links $(2,1) \rightarrow (2,2)$ and $(1,2) \rightarrow (2,2)$. If there is a fault on one of the links, node $(1,1)$ sends the packet towards the other link to avoid this link fault. While this method can handle these one-away link faults, it introduces additional hardware cost. Moreover, it must assume that only nodes can fail on negative edges, since it cannot handle link faults in these situations.

3.3 A Link-Fault-Tolerant Routing Algorithm

To improve the Glass/Ni routing algorithm, we propose a routing algorithm that is capable of handling any link faults in the network [33]. The idea is that the negative-first routing is preserved, but illegal turns are allowed. Potential deadlock is avoided by allowing a node to drop a packet to prevent the occurrence of cyclic deadlock. To guarantee one fault tolerance, all nodes must be able to communicate using one alternative route. This algorithm uses nonblocking illegal turns to handle the two cases mentioned above.

For example in the case showed in Figure 3.3, the packet is allowed to make the turn from east to south at node $(2,1)$. A deadlock may occur only when all the four links have been obtained by packets in a cycle as illustrated in Figure 3.3. This method though does

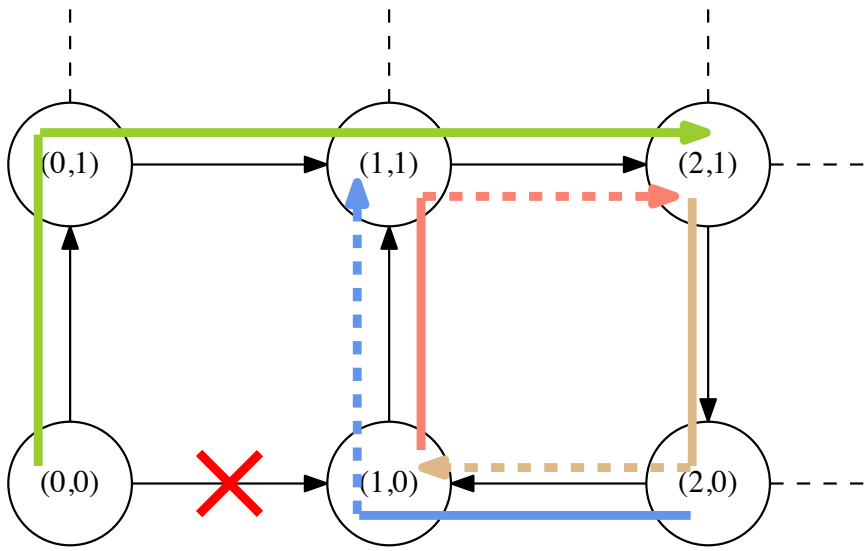


Figure 3.5: Packet is dropped by node 21 to break deadlock.

not block the packet moving from $(1,1)$ to $(2,0)$. Instead, if the link $(2,1) \rightarrow (2,0)$ does not clear in a sufficient time, the packet from $(1,1)$ is simply dropped by node $(2,1)$, effectively breaking the cycle,[1] as illustrated in Figure 3.5. To handle the one-away faults as illustrated in Figure 3.4, the improved routing algorithm allows illegal turns for every node in the network. For example, in Figure 3.6, a packet is delivered from node $(0,1)$ to node $(2,2)$ and encounters a fault on link $(2,1) \rightarrow (2,2)$. In this algorithm, the packet is redirected to node (3,1), and it follows the route $(3,1) \rightarrow (3,2) \rightarrow (2,2)$. This route requires an illegal turn on node $(3,2)$. However, if the link $(3,2) \rightarrow (2,2)$ does not become free in a sufficient amount of time, the packet is dropped at node $(3,2)$, avoiding the potential deadlock.

The behavioral model of the routing architecture presented in [33] includes the communication mechanism for packet transmission between nodes, the link that can become faulty at any time, and the route forwarding computation in each node. The communication mechanism employs an asynchronous handshaking protocol defined in the channel-level VHDL package [66]. Functions such as *send*, *receive*, and *probe*, introduced in Chapter 2, are used to enable communication of data over a channel. A VHDL process executes a *send* or *receive* function when it wishes to request communication over the channel. A

[1] A router's waiting time before dropping a packet can be determined at the implementation level.

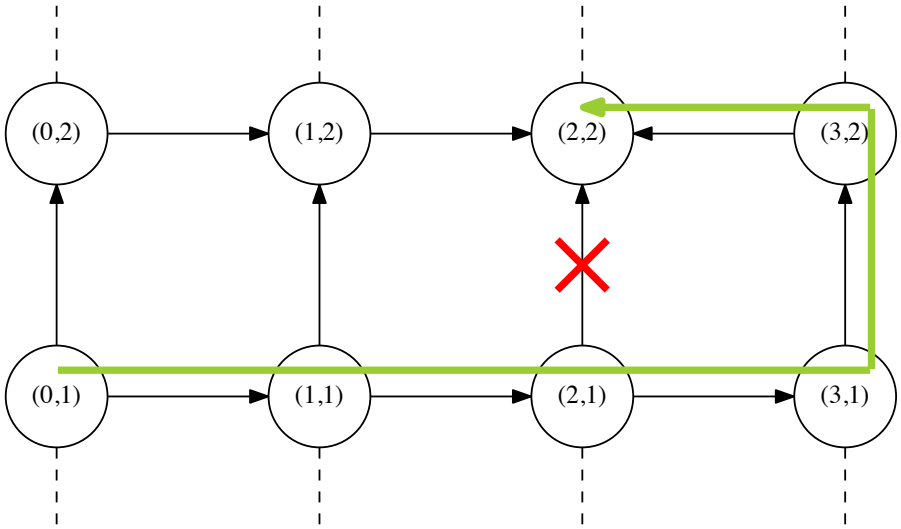


Figure 3.6: One-away fault handling example.

communication occurs when one process connected on one *port* of a channel requests to send while another process connected to the other port of the channel request to receive. The data is then transmitted over the channel using a handshaking protocol. The *probe* function is utilized to determine if the process connected on the other port of a channel has a pending request on the channel. Examples of channel communications are demonstrated through a series of producer-consumer examples in Chapter 2.

Each link shown in the 4-by-4 mesh in Figure 3.1a has a buffering capacity of one packet, which is modeled as a buffer shown in Figure 3.7. In order to model link failures, this buffer has a tunable probability of failure. In particular, this buffer actually includes two channels, a *Good* channel and a *Bad* channel. The *Channel select* signal is used to randomly choose for each packet transaction whether to use the *Good* or *Bad* channel. If the *Bad* channel is selected, the packet cannot be transmitted to the next routing node (i.e., the link is faulty). Otherwise, the link is functional and data is communicated over the *Good* channel to the *Buffer Out* channel to the next routing node. The probability of selecting between the two channels can be set to different values so that different levels of link reliability can be simulated. In order to use this faulty buffer, before sending out a packet, a routing node first uses the *probe* function to check whether the buffer at its output port is ready and able to accept a new packet. If the buffer is full, then neither the *Good* nor the *Bad* channel is ready. Once the buffer becomes empty, a random selection is made and either the *Good* or

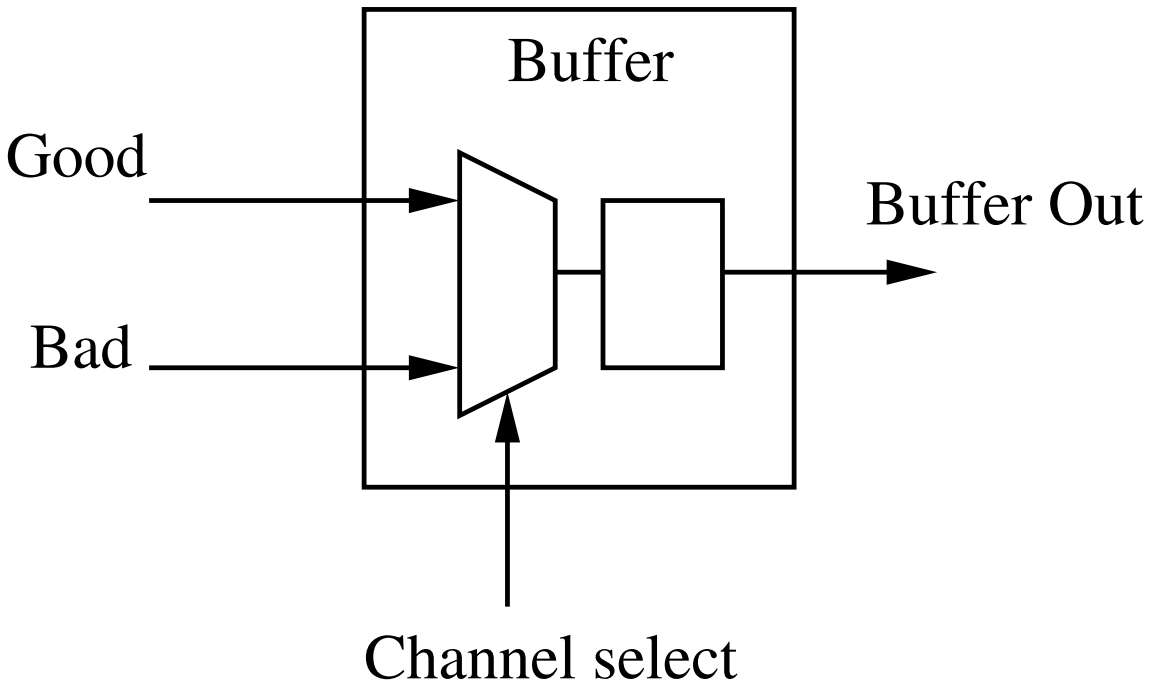


Figure 3.7: Link buffer with tunable probability of failure.

Bad channel becomes ready. If the *Good* channel is ready, the routing node can transmit the data to the buffer. When the *Bad* channel is ready, no data transmission occurs, and the routing node simply handshakes with the *Bad* channel to complete the communication, after which the buffer makes a new channel selection.

The routing algorithm works as follows. Each router communicates with its corresponding PE, and when a PE node (x, y) wishes to send a packet to another node (x', y'), it injects that packet into the network via its router. Based upon the intended destination of the packet, the router determines a direction to forward the packet. Following the Glass/Ni routing rule, each node first routes packets south and west, overshooting by one position from the destination in either direction, and then sends them north and east. One exception is that if the destination is one node away and on the east or north of the source node, i.e., $(x' = x + 1, y' = y)$ or $(x' = x, y' = y + 1)$, then the node sends such packets east or north first. After selecting a direction, the node attempts to communicate with the desired buffer on the link. At this point, one of three things can occur. First, the link may be busy, and the router must wait its turn to use the link. Second, the link may be faulty, and the node must either send the data in another direction or drop the packet if no viable alternative exists. Finally, the link may be functional and ready, and the node forwards the packet to the output buffer. The buffer then relays the packet to its successor node, which executes

the same algorithm. When receiving a packet from its neighboring node, a node may have to attempt an illegal turn to forward the packet. This is only successful if the link is functional and ready, otherwise the packet is dropped to avoid potential deadlock. Once a packet reaches its destination (x', y'), the packet is consumed. In summary, this link-fault-tolerant routing algorithm has the following rules for determining the next forwarding direction:

1. Route the packet east or north first if the destination is only one node away on the east or north, i.e., $(x' = x + 1, y' = y)$ or $(x' = x, y' = y + 1)$.

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

3. Route the packet east and north to the destination, avoiding routing the packet as far east or north as the destination for as long as possible.

4. Illegal turns are allowed only if making such a turn does not cause potential cycle of dependencies which leads to deadlock. Otherwise, the packet attempting the illegal turn must be dropped.

3.4 Evaluating Packet Loss Rate for Single-Fault Tolerance

To examine the importance of single link-fault tolerance, consider the 2-by-2 mesh network shown in Figure 3.8. Let us denote the probability of a link being faulty as p_f. By setting p_f to be the same for every buffer, faults are distributed randomly on the network. These faults are transient faults, since the fault state is determined for each individual transaction. For example, for every 1 million packets with an average of 5 hops, 50 thousand faults are injected when p_f is 1%. The network reliability can be characterized as the probability of a packet being dropped due to link faults, which is denoted by P_{fault}. Consider a packet that is injected at node $(0, 0)$ and destined for node $(1, 0)$. In a nonadaptive algorithm, only one choice of route is possible (i.e., $(0, 0) \rightarrow (1, 0)$), so $P_{fault} = p_f$. If two choices can be made at node $(0, 0)$ (i.e., $(0, 0) \rightarrow (1, 0)$ or $(0, 0) \rightarrow (0, 1) \rightarrow (1, 1) \rightarrow (1, 0)$), the probability is calculated as:

$$\begin{aligned} P_{fault} &= p_f^2 + p_f(1 - p_f)[p_f + (1 - p_f)p_f] \\ &= 3p_f^2 - 3p_f^3 + p_f^4 \\ &\approx 3p_f^2 \end{aligned}$$

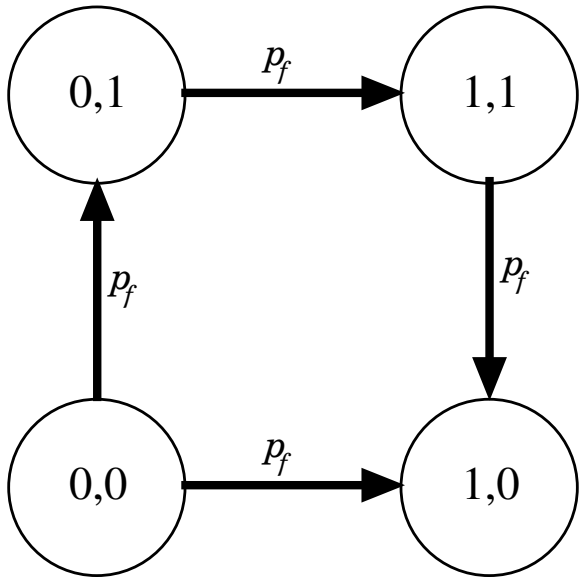


Figure 3.8: A 2-by-2 mesh network with fault probability of p_f on each link.

Note that the single link-fault probability is extremely rare, i.e., $p_f << 1$, so $3p_f^3$ and p_f^4 can be ignored. For node $(0,0)$'s two other possible destinations, node $(0,1)$ and node $(1,1)$, P_{fault} is still about $3p_f^2$. Although the Glass/Ni algorithm can tolerate single faults in most cases, this is not true in some special cases in the link-fault model, as mentioned previously. Therefore, the packet loss rate in the Glass/Ni algorithm is dominated by p_f, i.e., the probability of a packet being dropped is increased significantly, even if there is only one case where a single fault cannot be tolerated. If transient faults are randomly distributed in the network, these unsupported situations can easily be encountered and determine the overall packet loss rate. It is therefore necessary to maintain one-link-fault tolerance to keep a low packet loss rate.

This improved routing algorithm, however, may introduce additional packet loss due to deadlock avoidance when attempting to make illegal turns. Therefore, the overall packet loss rate, $P_{overall}$, is given as follows:

$$P_{overall} = P_{fault} + P_{deadlock},$$

where $P_{deadlock}$ is the additional packet loss due to deadlock avoidance. It is important to note that while P_{fault} is independent of the packet injection rate in the network, $P_{deadlock}$ is a function of this rate. Therefore, $P_{deadlock}$ can be minimized by limiting the network traffic load, adding additional link buffering capacity, or resending these dropped packets.

Since $P_{deadlock}$ is a function of packet injection rate, it is important to determine the injection rates of interest. For this purpose, a VHDL model of the link-fault-tolerant routing

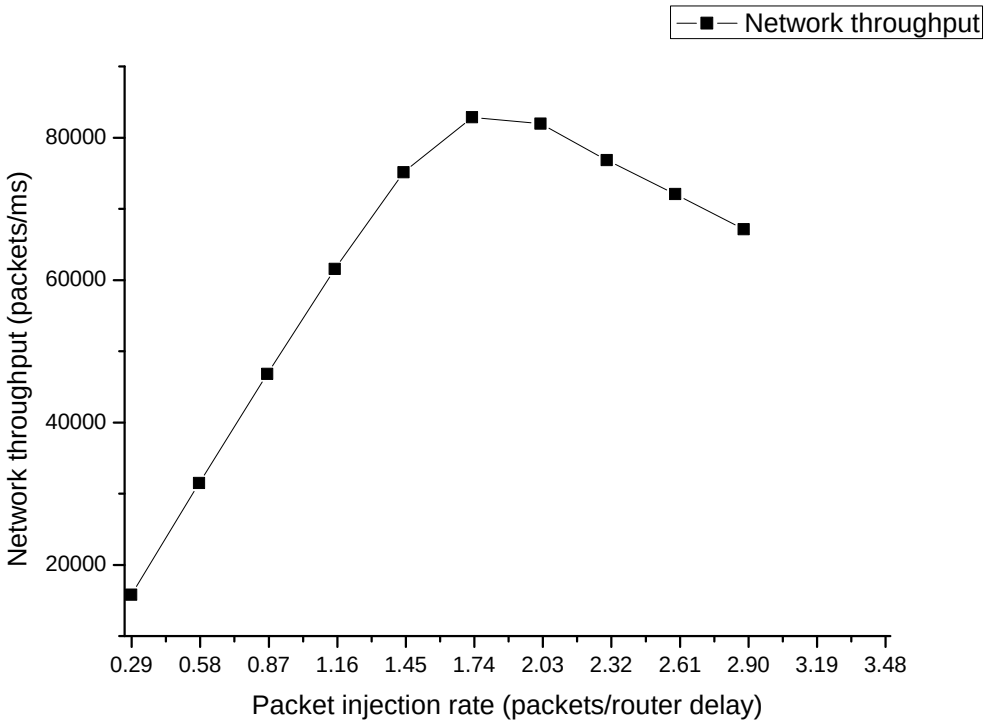


Figure 3.9: Network throughput vs. packet injection rate.

algorithm described previously was constructed and simulated on a 4-by-4 2D mesh. The link-fault probability is the same for all links in the mesh, and is modeled with the help of the *selection* function defined in the nondeterminism package (see Appendix A of [66]). For instance, the one percent link-fault probability is achieved by letting the selection function return an integer number uniformly drawn between 1 and 100, and the *Bad* channel becomes ready when this number is 100, otherwise the *Good* channel is set ready. Figure 3.9 shows the network's throughput under different packet injection rates. The packet injection rate is measured by the number of packets injected into the network per router delay. Note that all simulations are run for about 1000 ms, with an average of 80 million packets injected following a uniform traffic pattern. The plot shows that the network's throughput reaches its maximum at an injection rate of about 1.8 packets/router delay. After that, the throughput drops due to network saturation. Since the network only works efficiently when it is not saturated, our evaluations of fault-tolerance performance are focused on injection rates below about 2.0 packets/router delay.

Figure 3.10 presents a comparison of packet loss rates for the proposed link-fault-tolerant algorithm with a nonadaptive routing algorithm and a variation on the Glass/Ni algorithm

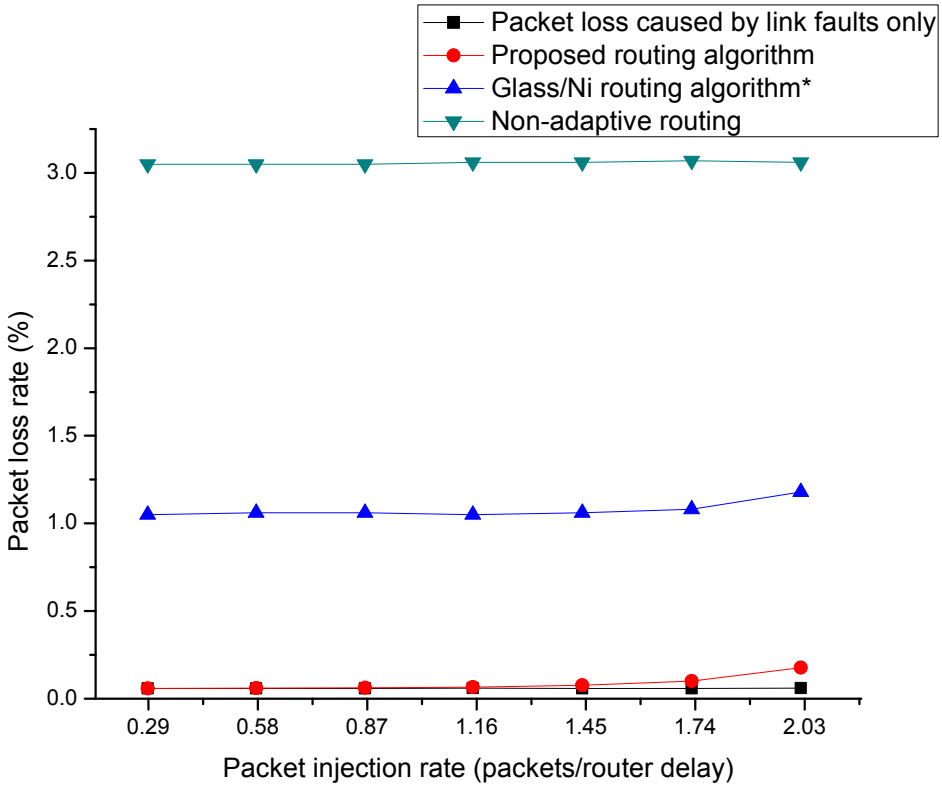


Figure 3.10: Packet loss rates for different routing algorithms.

that avoids the deadlock problems on the negative edges. This variation does not handle one-away link faults. The packet loss rate is measured as the percentage of dropped packets out of the total number of injected packets. These results demonstrate that the packet loss rate of the proposed routing algorithm is significantly better. Even though one-away faults are only part of all single fault cases, failure to handle these types of faults increases the packet loss rate by 20 times compared to our proposed algorithm. This justifies the previous analysis that the existence of any single link-fault problems ends up dominating the overall packet loss rate. Since our algorithm gives priority to deadlock avoidance, the trade-off is that packets get dropped when there is a potential for deadlock. To see this tradeoff, this figure includes both the overall packet loss rate as well as the loss rate due to link faults only. While for low packet injection rates most packet loss is due to link faults, as this rate increases, the packet loss due to deadlock avoidance begins to dominate.

3.5 NoC Architecture for Multiflit Wormhole Routing

Wormhole routing has been utilized in many NoC designs, such as Aethereal [100], Hermes [101], and QNoC [102]. It is a switching technique that routes a packet of data

in small units, known as *flits*. A packet travels through the network like a worm, and it typically consists of a *header flit* with the packet's destination, *body flits* carrying the packet's information, and a *tail flit* indicating the end of the packet. All flits of a packet take the path established by the header flit. The advantage of wormhole routing is the small buffer space in the switches that is needed and the pipelined data transfer.

The architecture of a node discussed so far simply consists of a PE and a router, as shown in Figure 3.1b. The router of a node, however, handles only one packet at a time, which is a limiting factor for the network throughput. This performance degradation can significantly worsen when each packet consists of multiple flits. Since the router of each node forwards one flit at a time, a multiflit packet may occupy the node for a significant amount of time, preventing other packets from using the same node. Therefore, the network can easily saturate when only a small number of such packets are in flight. To improve the throughput, a new architecture is introduced in [35] to allow each node to handle multiple multiflit packets simultaneously. For example, node 00 may be routing a packet from node 01 to node 10, while simultaneously routing a packet from node 10 to node 01.

The improved architecture for a three-by-three mesh is depicted in Figure 3.11. There are nine types of router nodes for this architecture. Namely, there are four types of corner nodes (i.e., nodes 00, 02, 20, and 22), four types of edge nodes (i.e., nodes 01, 10, 12, 21), and one type of center node (i.e., node 11). A larger network would include duplicates of the routers shown here. This architecture implements the extended version of the routing algorithm described in Section 3.3. To accomplish this, each node xy is composed of several independent *routers* (r_D_xy) and *arbiters* (arb_D_xy), where $D \in \{\mathrm{PE}, \mathrm{E}, \mathrm{N}, \mathrm{S}, \mathrm{W}\}$ corresponds to the direction the packet is coming from in the case of routers and going to in the case of arbiters.

The routing algorithm works as follows on this new architecture. Each node communicates with its corresponding *processing element* (PE), and when the PE of a node xy wishes to send a packet to the PE of another node $x'y'$, it injects that packet into the network via its *router* (r_PE_xy). Based upon the intended destination of the packet, a router determines a direction D to which to try to forward the packet, and then attempts to communicate with the *arbiter* (arb_D_xy) in charge of the corresponding link. At this point, one of three things can occur. First, the link may be busy, and the router must wait its turn to use the link. Second, the link may be faulty, and the router is instructed to find an alternate route. Finally, the link may be free, and the arbiter may nondeterministically select to communicate with this router over any other routers that may be trying to obtain this link.

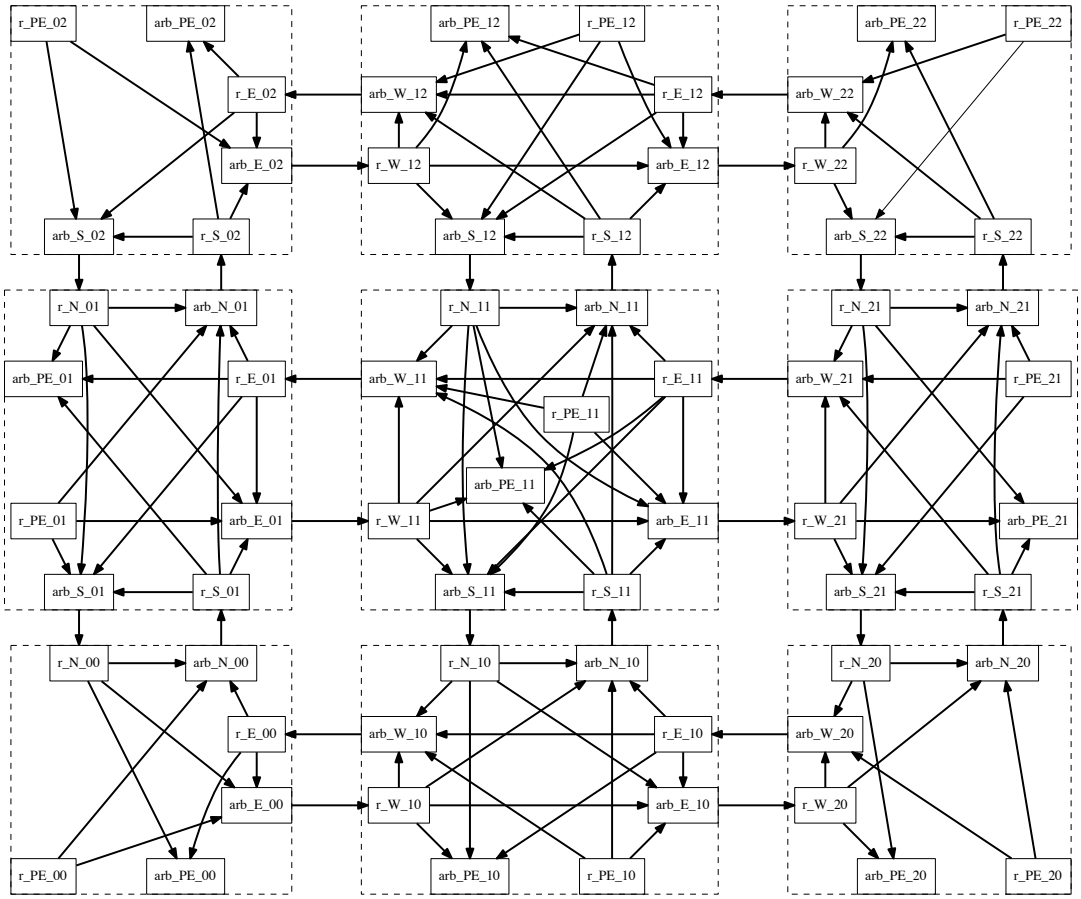


Figure 3.11: Architecture of the nine routing nodes in a three-by-three mesh.

The arbiter then forwards the packet one flit at a time to the succeeding router (i.e., the router the output of the arbiter is connected to), which then executes the same algorithm. Once a packet reaches its destination $x'y'$, the packet is consumed by the arbiter connected to its PE (arb_PE_$x'y'$).

Assuming there is at most one link-fault, an alternate route always exists, but it may require an illegal turn. For example, consider the two-by-two mesh shown in Figure 3.12 and assume that node 10 wishes to send a packet to node 01. In this case, a west then north route is the preferred option. If arb_W_10 reports a fault on its link to r_E_00, r_PE_10 must communicate with arb_N_10 instead. Once the packet reaches r_S_11, this router must make an illegal turn and route the packet west through arb_W_11. However, arb_W_11 may be busy routing a packet from node 11 to node 00. This packet in turn may be blocked because arb_S_01 is busy routing a packet from node 01 to node 10. Similarly,

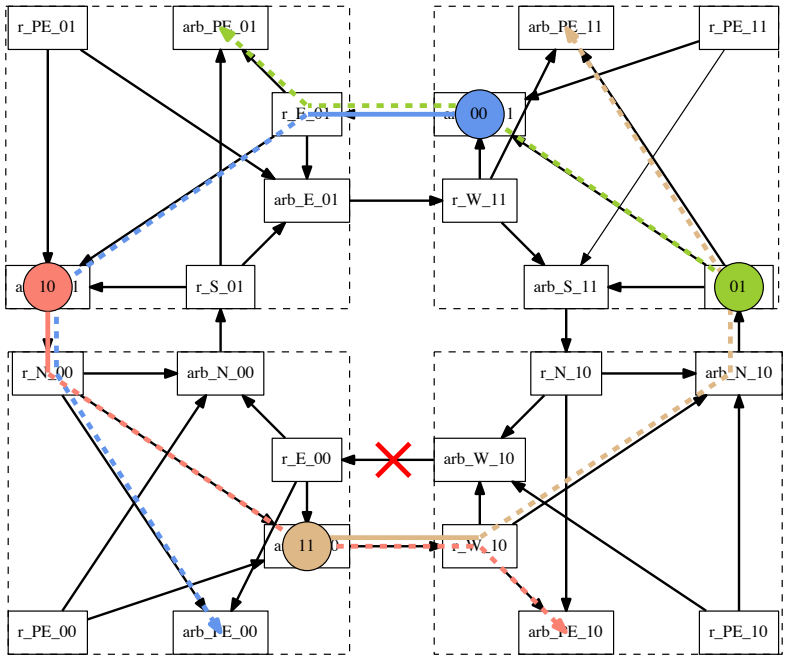


Figure 3.12: Illustration of a deadlock caused by a cyclic communication dependency. Each packet is represented by a circle containing its destination. A solid line coming out of a packet indicates the router where it is currently traveling to. A dashed line indicates the remaining route of a packet. The crossed link indicates that the link from node 10 to node 00 is faulty.

this packet may be blocked because arb_E_00 is busy routing a packet from node 00 to node 11. Finally, this packet is blocked because arb_N_10 is busy due to the packet from node 10 to node 01. Taken all together, there is a communication cycle causing a deadlock, as is illustrated in Figure 3.12. In this case, arb_W_11 sends a *negative acknowledgement* to r_S_11, indicating its unavailability to accept a packet from this router, which tells this router (r_S_11) to drop the incoming packet, removing the communication cycle and avoiding the potential deadlock.

3.6 Conclusion and Discussion

This chapter introduces the Glass/Ni fault-tolerant routing algorithm on a 2D-mesh topology. Its node-fault assumption, which treats a faulty link as a node fault, severely limits its practical application. Switching to a link-fault model, however, fails to handle both cyclic deadlock and one-away faults. Next, an improved routing algorithm with one link-fault tolerance is described, with additional packet loss as a tradeoff. This cost, however,

is shown to be minimal from the simulation results of a VHDL implementation of the routing algorithm. An extended routing architecture is then introduced to enable simultaneous routing of multiple packets on a single node, and improve the efficiency of multiflit wormhole routing.

Note that the deadlock avoidance mechanism used by the link-fault routing algorithm detects potential deadlock in a conservative way. If the link on the way of an illegal turn is not available, i.e., either busy or faulty, a packet making this turn gets dropped. However, such cases do not necessarily mean deadlock can occur due to the lack of cyclic dependency, as it may be the case that it is the only busy link. Also, deadlock detection is based on a one-time probing execution on a communication channel. Depending on the execution time, the result may vary, which consequently alters the decision for packet drop. For instance, the link on the illegal turn may be occupied when the probe is executed, but after it quickly clears, probing it again gives a different result that prevents the packet drop. It is possible to execute multiple probe operations for some amount of time before a packet is dropped. This modification can improve the accuracy of deadlock detection, but it requires more complicated methods which, in turn, require additional hardware.

The presented link-fault-tolerant routing protocol in this chapter is formally modeled using a process-algebra action-based approach in Chapter 4 and a LPN state-based approach in Chapter 5. These chapters also describe detailed verification results of several desired functional properties for this routing algorithm.

CHAPTER 4

FORMAL ANALYSIS USING CADP

This chapter first describes several key lessons that are learned during the evolution of the LNT model of the NoC architecture introduced in Section 3.5. Formal analysis of several important diagnostic examples reveals design flaws leading to false behaviors such as unexpected packet drop and deadlock in the routing architecture. With the help of a data abstraction technique, deadlock freedom and single-link-fault tolerance are verified. In order to verify packet delivery, this chapter then introduces a refined data abstraction technique. This refinement leads to the discovery of a potential livelock problem through formal analysis on the link-fault tolerant NoC architecture presented in Section 3.5 of Chapter 3. In the process of eliminating this problem, an improved routing algorithm is derived. The improvement simplifies the routing architecture, enabling successful verification using the CADP verification toolbox [103]. The routing algorithm is proven to have several desirable properties, including deadlock and livelock freedom, and tolerance to a single-link-fault. Finally, this chapter describes several remaining challenges to the verification of this and similar systems.

4.1 Background and Related Work

Verification of a concurrent system using a process algebra approach typically performs either *equivalence checking* or model checking. Equivalence checking tests behavioral equivalence between two models: one detailed that is close to the actual implementation of a system, and one abstract that describes the sequences or trees of relevant actions the system has to perform. Model checking proves whether a system specification, described as process terms, satisfies temporal logic formulae. It is the model checking approach that is of interest for the NoC verification in this chapter. In particular, our approach constructs a system's LTS in a compositional fashion where LTSs for each component of the system are generated first, and are incrementally composed and minimized to obtain the LTS of the complete system. Minimization steps on a LTS are governed by a behavioral equivalence specification.

In many situations, the interest of a system is its behavior with respect to the outside

world: its reaction to the environment's stimuli and its effect on the environment. Behavioral equivalences are therefore mostly defined as relations that describe two indistinguishable labeled transition systems under some external observations, e.g., experiments or tests. They allow one to replace a LTS with an abstract, equivalent counterpart with a smaller state space that omits unwanted details. A state minimization technique typically defines a process of obtaining such an abstract replacement. Combined with the step-wise incremental composition, state minimization plays a key role in maintaining manageable intermediate state size before the state space for the complete system is generated.

Over the years, the need of analyzing different properties has promoted propositions of many theories of equivalences in the literature. A comprehensive review of major classes of equivalences is described in [104]. The main families of equivalence relations over two systems' LTSs are based on *traces*, *decorated-traces*, and *bisimiluation*. Traces equivalences require two systems to execute the same sequences of actions, and decorated-traces adds an additional requirement on equivalent states after each sequence: they are ready to accept the same *set* of actions. Bisimulation-based equivalences (or *bisimilarity*) define the equivalence relations recursively in a stepwise fashion on each pair of equivalent states: two states are bisimilar if they reach states that are also bisimilar via the same action.

A classical vending machine example shown in Figure 4.1 illustrates different levels of granularity of these equivalence relations. The trace equivalences consider all of them as equivalent, while the bisimulation-based equivalences distinguish all of them. The decorated-traces equivalences are able to distinguish the LTS in Figure 4.1a from the other two, but consider the two LTSs in Figures 4.1b and 4.1c equivalent. This is because state r_2 is ready to execute either action *coffee* or *tea*, but no such equivalent state can be found on either of the other two machines. However, every state of machine in Figure 4.1b can find an equivalent state on that of Figure 4.1c, and vice versa. Bisimilarity identifies the difference between the first and the second machine after the insertion of the second coin: the first machine still offers the user a choice of hot beverages while the second one does not. Similarly, it distinguishes the second one from the third one after they accept the first coin, at which point the third machine already takes away the user's choice.

Despite the strong power of bisimilarity to distinguish system's behaviors, in some cases, fine-grained discrepancies in their behaviors are of little interests to an external observer. For example, in Figure 4.1, a user cannot tell the difference between the second and the third vending machines by observing the results of two coin insertions, as they both disallow the user's choice of beverages. To address equivalences based on external observations, *testing*

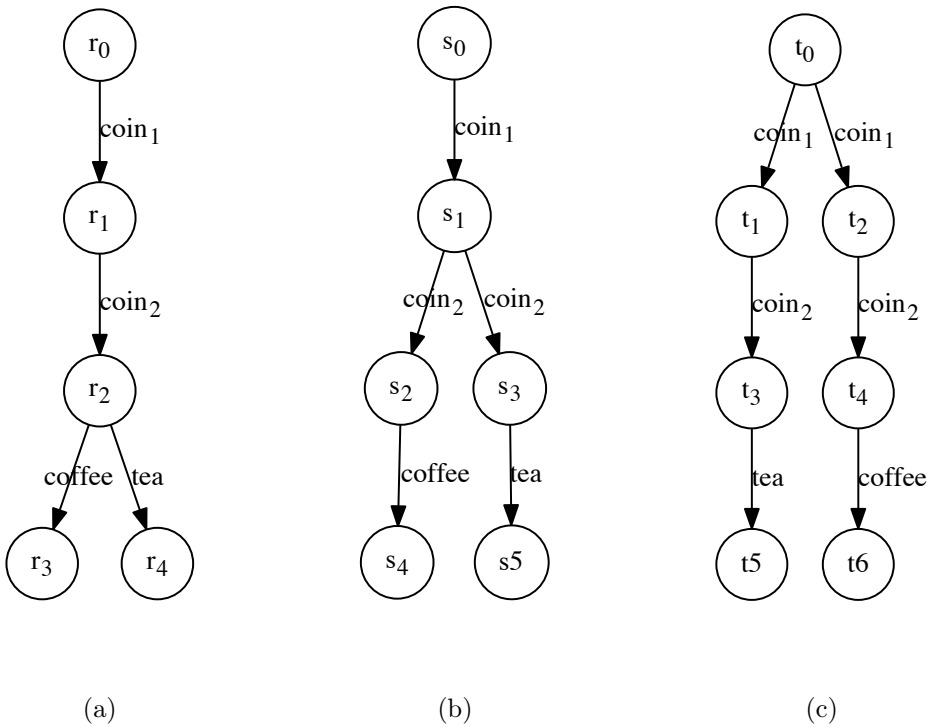
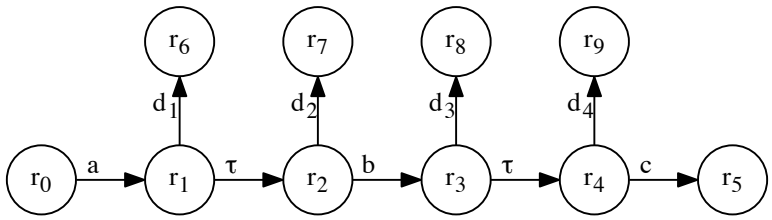


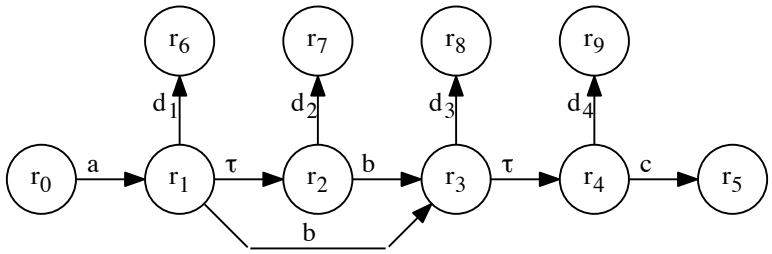
Figure 4.1: Three vending machines.

equivalence was proposed in [105], which does not differentiate systems that cannot be taken apart by external observers. Two systems are test-equivalent if they lead to successful observations by the same sets of observers.

Weak variants of the aforementioned equivalences have been proposed to define equivalences for LTSs with *internal actions*, i.e., actions that are not observable. *Weak bisimulation* requires that every visible action in one LTS corresponds to the same visible action in the other, possibly preceded or followed by an arbitrarily long sequence of internal actions, and every internal transition in one LTS should correspond to an arbitrarily long (possibly empty) sequence of internal actions. The major shortcoming of this equivalence is that it does not preserve the branching structure of processes and hence lacks one significant characteristics of bisimulation semantics. As an example, consider two LTSs in Figure 4.2. They are equivalent under weak bisimulation because the introduced visible action b from state r_2 in Figure 4.2b does not violate this equivalence between the two LTSs. However, this added action introduces a computation sequence that ignores the execution of action d_2 from the initial state r_0. To preserve branching structures, an alternative equivalence, the *branching bisimulation* equivalence (or *branching bisimilarity*) [106], has been proposed to consider



(a)

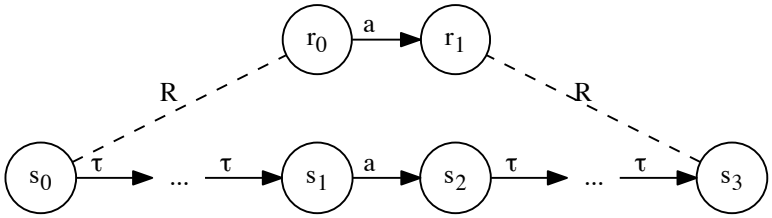


(b)

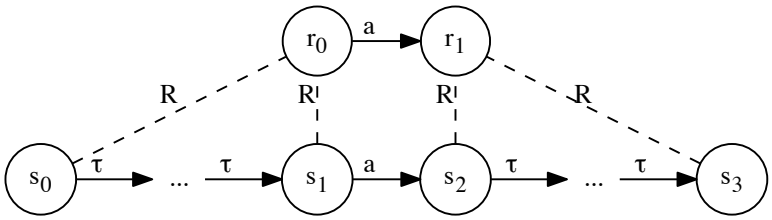
Figure 4.2: Weak bisimulation equivalence.

equivalences of internal actions that appear at the corresponding branching points of two LTSs. Added upon the equivalence relation for weak bisimilarity, branching bisimulation equivalence requires that the states of an internal sequence in one LTS are related to the *same* state in the other. The differences between branching and weak bisimulation are characterized in Figure 4.3. The starting and end points, i.e., s_0 and s_1, of the internal sequence path are required to be equivalent to r_0 for branching bisimulation, but only s_0 is required to be equivalent to r_0 for weak bisimulation. It is obvious that the two LTSs in Figure 4.2 are not branching bisimilar, because in order to find an equivalent sequence segment for $r_1 \xrightarrow{b} r_3$ of Figure 4.2b, it has to be the case that r_1 and r_2 in Figure 4.2a both correspond to r_1 in Figure 4.2b, which is not the case.

The *divergence-sensitive branching bisimulation* equivalence adds on top of branching bisimilarity the condition that checks for equivalence of the corresponding *τ-loops* in two LTSs. A τ-loop is an internal action sequence that starts from and returns to its initial state. A practical meaning of a τ-loop is the existence of *livelock*. This equivalence relation is of interest to us as it preserves deadlocks, livelocks, linear-time, and branching-time



(a) weak bisimulation



(b) branching bisimulation

Figure 4.3: Differences between weak and branching bisimulation.

properties [107]. Compared to branching bisimulation that collapses every livelock into a deadlock, divergence-sensitive branching bisimulation preserves livelocks and therefore can reveal true deadlock scenarios.

There have been several previous works that have applied model checking to NoC routing algorithms. For example, to facilitate the use of model-checking techniques, automatic translations are developed from the asynchronous hardware description language CPH (*Communicating Hardware Processes*) to networks of automata [108] and to the process-algebraic language LOTOS [70]. The latter approach is applied to verify an input controller [109] for an asynchronous NoC [110] that implements a deadlock-free routing algorithm based on the odd-even turn model [99]. However, this NoC does not handle failures. Deterministic XY routing algorithms, whose routing logic are significantly simpler than the fault-tolerant routing algorithm presented in this chapter, have been previously studied [111, 112], leading to the verification of functional properties requiring little network traffic, such as packet delivery. Also, Chen et al. face state explosion when attempting to verify deadlock freedom [111], and Palaniveloo and Sowmya mention no results on deadlock verification [112]. Lugan et al. verified an *optical* NoC with four initiators and four targets using Uppaal [113]. To reduce

the verification time, their verification used a two-level approach (a first verification on an abstract level, complemented by a more detailed verification of a part of the NoC). Their NoC, however, is not fault-tolerant, and it has highly symmetric processes.

An interesting alternative to model checking is to use static analysis. Verbeek and Schmaltz [114] proposed a necessary and sufficient condition for deadlock-free wormhole routing that can be statically computed independently from the network status. This condition is used in a decision procedure for deadlock detection on large networks from a wide range of NoC topologies and routing algorithms [115]. With the help of the Dci2 (*Deadlock Checker In Designs of Communication Interconnects*) tool [116], Alhussien et al. [117] proved deadlock-freeness, livelock-freeness and packet delivery of a fault-tolerant wormhole routing logic for large-scale mesh networks. A formal NoC specification and validation environment, GeNoC (*Generic Network-on-Chip*), implemented in the ACL2 theorem prover, was first proposed by Borrione et al. [118]. It was used to verify a nonminimal adaptive routing algorithm in [119]. An improvement of the GeNoC model [120] is proposed to enable static verification of deadlock freedom and livelock freedom, as well as functional correctness. It was shown to prove these properties on an adaptive west-first routing algorithm on a Hermes NoC, with approximately 86 percent of the proof automatically derived. By using static analysis, both the Dci2 and GeNoC approaches are extremely efficient for checking the same properties checked in this chapter for deadlock prevention routing algorithms. This efficiency enables them to be applied to large networks. These approaches, however, are incapable of verifying properties of deadlock avoidance algorithms, as this requires a dynamic analysis.

4.2 Evolution of Formal NoC Models

This section describes several key lessons learned in the process of developing a Lnt model of the two-by-two mesh shown in Figure 4.4. Formal analysis of the diagnostic information on the examples in this section has revealed flaws in our routing architecture that are challenging to detect otherwise. A direct verification approach consists in generating the corresponding Lts. If successful, the generated Lts can be used to analyze functional properties of the protocol. The Cadp toolbox supports compositional techniques to alleviate the exponential growth of the number of states. In a nutshell, compositional Lts generation proceeds in a "bottom-up" manner, starting with individual processes and alternating generation and minimization steps. Svl script automates the compositional Lts generation, implementing heuristics [121] to optimize the order processes.

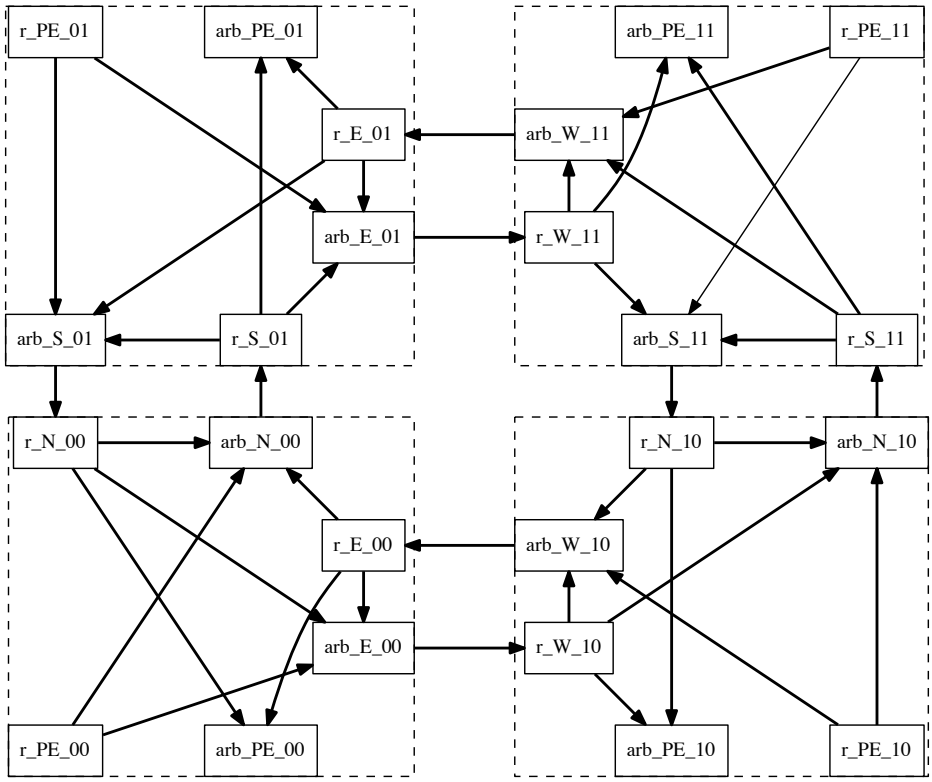


Figure 4.4: Architecture of the two-by-two mesh.

4.2.1 One Direction Routing

The first model developed and verified is the simple one-direction routing model shown in Figure 4.5. It is advantageous to construct a model with one complete cycle consisting of partial components from each node, since the model is simple enough for testing asynchronous communications between any two components. Also, the resultant state space is manageable, enabling the efficient checking for deadlock and packet loss without having to abstract the model. Since this model only has one routing direction, there are no alternative routes available, avoiding the need to model route-forwarding computation in each router. Having only the counterclockwise routing direction forces the north-to-west illegal turn to occur on the northeast node.

In this first model, each PE router only generates one single-flit packet destined to the node in its diagonal direction. For example, the PE connected to node 01 sends a packet to node 10. After emitting one packet, each PE router becomes inactive. No components consume any packets and it is assumed that no link fault exists in the network. The expected

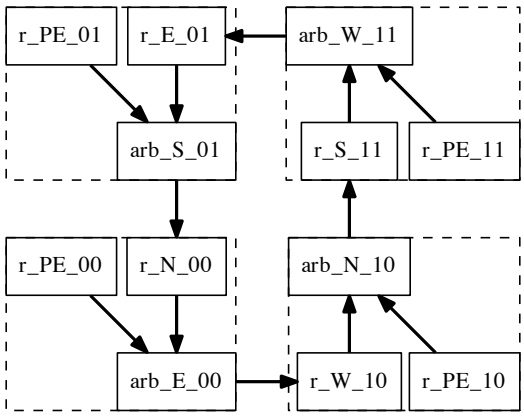


Figure 4.5: A counterclockwise routing model.

behavior is that the packet from node 10 to node 01 gets dropped due to deadlock avoidance, and the remaining three packets keep cycling through nodes in the network forever, and no deadlock exists. The arbiter, arb_W_11, on the northeast corner is responsible for detecting the potential deadlock by checking availability of its succeeding router r_E_01. To avoid deadlock, it informs its preceding router r_S_11 to drop the packet when router r_E_01 is not available.

The LNT descriptions of arb_W_11 and r_S_11 are shown in Figure 4.6. The arbiter arb_W_11 is modeled by the LNT process "*arbiter_nack*". The three gates "PEr_Wa_11", "Sr_Wa_11", and "n11_n01" (between square brackets) of this process correspond to the three links (r_PE_11 → arb_W_11), (r_S_11 → arb_W_11), and (arb_W_11 → r_E_01) in Figure 4.5, respectively. The contents of each flit is represented by a natural number, and the arbiter process uses the variable "one_flit" of type "Nat" to store the flit travelling through it.

The behavior of this process is a nonterminating loop with two nested choices. The outer choice decides whether the arbiter is ready to receive a packet: if its preceding router r_E_01 confirms its availability by synchronizing on the "n11_n01" gate with the arbiter, then it starts to receive the packet; or the arbiter issues a negative acknowledgement "Sr_Wa_11(**false**)" to r_S_11 indicating that its output is blocked by another packet. If both options are available, one is chosen nondeterministically. When the arbiter is ready to receive a packet, it nondeterministically chooses between the PE router, r_PE_11, and the south router, r_S_01. Since taking a packet from r_S_01 effectively makes an illegal

```
process arbiter_nack [PEr_Wa_11, Sr_Wa_11, n11_n01 : any] is
var one_flit : Nat in loop
  select
     n11_n01; -- Router r_E_01 is ready to accept packet
     select
        PEr_Wa_11(?one_flit); -- Receive packet from r_PE_11
        n11_n01(one_flit) -- Send packet to r_E_01
     []
        Sr_Wa_11(true); -- Send positive ack. to r_S_11
        Sr_Wa_11(?one_flit); -- Receive packet from r_S_11
        n11_n01(one_flit) -- Send packet to r_E_01
     end select
  []
     Sr_Wa_11(false) -- Send negative ack. to r_S_11
  end select
end loop end var end process

process router_drop_pkt [n10_n11, Sr_Wa_11 : any] is
var status : Bool, one_flit : Nat in loop
  n10_n11; -- Ready to accept packet from arb_N_10
  n10_n11(?one_flit); -- Receive packet from arb_N_10
  Sr_Wa_11(?status); -- Request arb_W_11's status
  -- Send packet to arb_W_11 ONLY on TRUE status
  if status then
    Sr_Wa_11(one_flit)
  end if
end loop end var end process
```

Figure 4.6: The LNT processes for arb_W_11 and r_S_11.

turn, this arbiter first sends an acknowledgement "Sr_Wa_11(**true**)" to r_S_11. The router r_S_11 (represented by the LNT process "*router_drop_pkt*") checks the received status of arb_W_11, and a false status leads to a packet drop: r_S_11 is ready to receive the next packet which overwrites the current one that needs to be dropped.

The generated state space for the counterclockwise routing model contains terminal states, indicating deadlocks in the model. Analysis of the diagnostic sequences of transitions reveals that all four packets can get dropped by the r_S_11 router, which is an unexpected behavior. According to the routing protocol, r_S_11 should drop a packet when arb_W_11 returns a false status, and the arbiter should do so *only* when its output, r_E_01, is busy serving other packets. As mentioned previously, it is possible that one packet is dropped for this reason, but the remaining three should stay in the network as the network is not congested anymore. Analysis of the outer choice on the arbiter's specification shows that there always exists a path where it sends a negative acknowledgement to tell the router r_S_11 to drop its packet. The nondeterministic choice enables sending both a true and a false acknowledgement to the router, and as long as the gate rendezvous for sending a false acknowledgement is possible, it gets a chance to occur. Therefore, arb_W_11 can always send a false acknowledgment regardless of potential deadlocks.

One possible improvement is to have a prioritized choice: the option of sending a positive

acknowledgement is always the preferred one. Ideally, availability of the preferred positive option should prevent the option of sending the negative acknowledgement. To implement this priority would require that the "**select**" operator could probe the possibility of a gate rendezvous on the preferred choice. Implementing the priority choice in LNT requires additional processes [70], which may lead to state explosion. Even if it can be verified, the packet leakage path may not get removed due to the timing of when the probes are executed. Another option is to prune the unwanted execution paths from the generated state space using the priority operator in EXP.OPEN/SVL [56]. However, the state space of the entire model is generated compositionally using branching bisimulation, which is not a congruence for the priority operator [122].

4.2.2 Removing Arbiter's Buffering Ability

Our initial analysis has focused on a two-by-two NOC shown in Figure 4.4, since state space generation for the three-by-three NOC even using compositional techniques is challenging. The intermediate state space corresponding to only 13 out of the 66 components in the three-by-three NOC already has several hundred million states. Including just one more component to construct the next intermediate LTS almost doubles the size of the LTS. Since there are still 52 components to be included, it is clear that this *growth* of the intermediate state spaces is unmanageable. Although it might be surprising, these large state spaces can be explained by the fact that each of the 66 components can store a packet (or be empty), resulting in a theoretical state space size of more than 10^{14} states for 14 components of the three-by-three NOC. Although the content of a packet is abstracted to just its destination coordinates, which are necessary to precisely determine a packet's next forwarding direction, the existence of many possible data values further contributes to the combinatorial state explosion.

One improvement to alleviate the state explosion problem is to reduce gate rendezvous between arbiters and routers in each node. This means that on the LNT model level, routers and arbiters in one node are merged into one process, removing the need for gate rendezvous between them. The resultant northwest routing node has the following behavior. It nondeterministically selects one among the following three operations: generating its own packet, receiving a packet from the northeast node, or receiving one from the southeast node. Once the node has a packet, based on the packet's destination, it attempts to send out the packet to the first choice of route, and tries alternative routes if the first one is not available. All the other three nodes have a similar behavior.

This simplification of the routing nodes indeed helps to reduce the state space. However,

it removes the buffering capacity in each arbiter, and consequently causes deadlocks. A typical deadlock scenario is that initially the four routing nodes generate their own packets at the same time, between the northwest and southwest nodes, and between the northeast and southeast nodes, the packet in one node tries to go the other. No nodes can make progress in this situation. To send a packet, a node needs its neighboring node to communicate on the same gate. It is required because the node's own arbiter, which connects to the neighboring node, cannot store anything, and only the neighboring node has the storing capacity. In the mean time, If the neighboring node is trying to do the same thing to this node, neither one can deliver packets because both are waiting for the other to accept their own packets. Removal of the arbiter's buffering ability also makes it impossible for one node to have multiple packets passing through it at the same time.

From this experiment, we conclude that arbiters in the network need storage capacity in order to relay a packet, freeing up their corresponding routing nodes to handle other communications. It also implies that simplifications on the node architecture without modifying the routing algorithm can introduce deadlocks in the system behavior. Therefore, removing interleavings of gate rendezvous on the LNT models is unsuccessful. [1]

4.2.3 Finding Data Abstractions

As mentioned earlier, a major contributor to the large state space is the existence of many data values in the model. The previous experiments do not consider data abstraction of a packet's content, because a router requires the packet's destination to decide its next forwarding direction: it is impossible for a router to perform routing computation without the destination information, although this information is only needed by the routers. In theory, all except the PE routers can receive packets destined to all node locations. Since our link-fault-tolerant routing algorithm allows illegal turns (cf. Section 3.5), it means that a router may potentially direct packets to all of its viable directions. The idea is, thus, to abstract the routing decision with nondeterministic choice. In other words, after receiving a packet, a router nondeterministically selects either its own node, indicating that the packet has reached its destination, or one of the (two, three, or four) forwarding directions for the packet, without the need to examine the packet's destination. This abstraction enables us to eliminate a packet's destination information. Moreover, since every packet is provided

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

with a preferred route and at least one alternative route for the purpose of fault-tolerance, the router's model should provide, in the nondeterministic choice, the possibility for every forwarding direction as the preferred route for a randomly destined packet, assuming the router does not perform an illegal turn. From the analysis of the routing algorithm, it is obvious that making an illegal turn is never a preferred choice for a route unless all forwarding routes of a router are illegal. For example, the north-to-east legal turn is always a preferred choice over the illegal north-to-south turn at router r_S_01 in Figure 4.4.

In a two-by-two mesh, there are three types of routers. First, there are routers r_S_11 and r_W_11 that can make two illegal turns (RI2). Next, there are routers r_W_10 and r_S_01 which can make one illegal turn (RI1). Finally, there are all other routers which never make illegal turns (RI0). Since routers in each of the three categories have the same abstract behavior, the rest of this section uses representatives, i.e., RI2, RI1, and RI0, to refer to routers in each category.

The next question is whether packets need to be modeled at all. Our first experiment shows that the model without packet information exhibits the packet leakage problem. As discussed in Section 4.2.1, the reason is an intrinsic feature of RI2, which has a nondeterministic choice of where to send a packet: either to RI2 itself or to an illegal forwarding direction. Without any packet information, taking an illegal forwarding direction is always possible, regardless of deadlock avoidance, effectively creating a leakage path. To fix this problem, an abstraction of a packet has to be included such that an illegal turn in RI2 is not always possible. An important feature of the routing pattern is that a packet takes an illegal turn only after its attempt to the preferred route fails due to a failure on the route. In other words, when a packet makes an illegal turn, it must have been diverted at least once before. Thus, a packet can be modeled as a single-bit Boolean variable, indicating its diversion status. In the RI2's LNT process "*router_two_illegal*" shown in Figure 4.7, only a diverted packet can take illegal turns. The "only if" statement is useful for implementing guarded commands. This restriction rules out the possibility of dropping a nondiverted packet, at which point the process terminates as all dropped packets should be diverted at least once before. Comparing to the precise routing decision, a packet with one forwarding direction in the concrete model has the possibility to be forwarded to any direction in the abstract model. Therefore, the abstraction is conservative in that it preserves all transition sequences in its corresponding concrete model. One subtle difference introduced in the abstract model is the notion of a diverted packet, which does not exist in the concrete model. It is, however, a feature that implicitly exists in the concrete model's routing behavior.

```
process router_two_illegal [input, out_arb_PE, out1_illegal,
                            out2_illegal, drop : any] is
var one_flit, arb_status : Bool in loop
  input(?one_flit);
  select
     out_arb_PE(one_flit)
  []
     only if one_flit == true then -- packet is diverted
       -- first try out1_illegal, then out2_illegal
       out1_illegal(?arb_status);
       if arb_status == true then
         out1_illegal(one_flit)
       else
         out2_illegal(?arb_status);
         if arb_status == true then
           out2_illegal(one_flit)
         else
           drop -- both illegal turns impossible
         end if
       end if
     end if
  end select
end loop end var end process
```

Figure 4.7: The LNT process for the RI2 router.

There are also three categories of arbiters, corresponding to the router categories. Figure 4.8 shows the arbiter corresponding to RI2. It selects between its PE router and two routers, from which flits may just have made an illegal turn. For each option the arbiter takes, after receiving a flit, it keeps rejecting requests from RI2 routers until it delivers the flit. When receiving rejections on all its illegal forwarding routes, RI2 drops the packet to prevent potential deadlock. The complete LNT specification for the two-by-two NoC is available at `http://www.async.ece.utah.edu/~zhangz/research/lnt_modeling/`.

4.3 Removing Livelock to Improve Routing

Although this data abstraction enables successful verification of properties like deadlock freedom and single-link-fault tolerance as described in [35], proving packet delivery is impossible, since it is always possible that some, if not all, packets produced get continuously dropped in the network. Using the available information in an abstract packet, one cannot know if a packet reaches its destination or gets dropped by a router on the way. In order to check packet delivery while keeping intermediate state spaces manageable, we investigated a hybrid modeling scheme, combining concrete and abstract packets. The idea is that for one experiment, one node generates a single concrete packet followed by repeated abstract packets, while all other nodes only generate abstract packets. In this way, the delivery of a particular concrete packet can be checked with the existence of abstract packets to model network traffic. All routers are modified to handle both types of packets. A router

```
process arbiter_nack_2 [in_PE_router, in1_illegal,
                        in2_illegal, arb_out : any] is
var one_flit : Bool in loop
  select
     in_PE_router(true); in_PE_router(?one_flit);
     loop L1 in select
        arb_out(one_flit); break L1
     []
        in1_illegal(false)
     []
        in2_illegal(false)
     end select end loop -- L1
  []
     in1_illegal(true); in1_illegal(?one_flit);
     loop L2 in select
        arb_out(one_flit); break L2
     []
        in1_illegal(false)
     []
        in2_illegal(false)
     end select end loop -- L2
  []
     in2_illegal(true); in2_illegal(?one_flit);
     loop L3 in select
        arb_out(one_flit); break L3
     []
        in1_illegal(false)
     []
        in2_illegal(false)
     end select end loop -- L3
  end select
end loop end var end process
```

Figure 4.8: The LNT process for the arbiter corresponding to RI2.

determines the packet's next forwarding direction either precisely based on its destination coordinates or nondeterministically if the packet is abstract. We can then exhaustively run all experiments for every possible concrete packet produced by all four nodes, combined with all possible single-fault locations, and check packet delivery properties on the LTS generated for each experiment.

4.3.1 Potential Livelock Problem

Consider the situation shown in Figure 4.9 with a faulty link, namely the output of arb_W_10. Router r_PE_10 of node 10 generates a concrete packet destined for node 01. The generated LTS for the concrete model shows that router r_N_10 might fail to find a route for this packet. However, this failure does not exist in the generated LTS for the corresponding abstract model [35]. This mismatch indicates that the abstraction for this router is not correct. The issue is that the fault-tolerant concrete routing logic for r_N_10 provides only one forwarding direction (W) and the only alternative route (N) is forbidden. No routing occurs if the only available route is faulty. However, the corresponding abstract

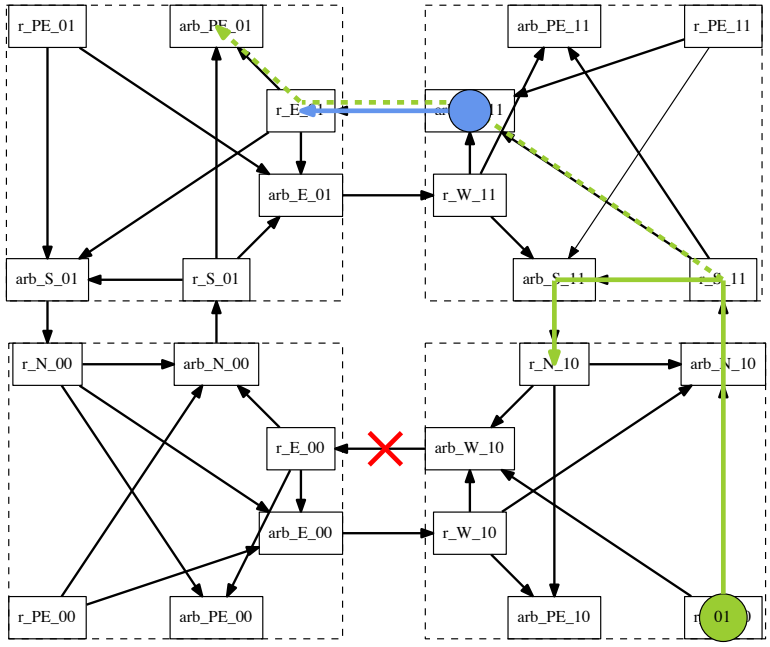


Figure 4.9: Illustration of the problem with the abstract model. The crossed link indicates that link from node 10 to node 00 is faulty. Solid thick arrows end in the routers' handling packets. The dashed arrow indicates the route for the packet from node 10 to node 01, taking into account the failed link.

routing behavior still provides the second routing choice, avoiding the routing failure when the first choice is not available. Below, we discuss this problematic example in detail to illustrate the incorrect abstraction and show how it should be fixed.

For a concrete packet to node 01, r_PE_10 first attempts to send it west to arb_W_10, but fails because of the faulty link. Then, it diverts the packet to send it north to r_S_11, which then diverts the packet south to r_N_10 because its first preference to the west is blocked (as arb_W_11 is serving another packet). Receiving a concrete packet destined for node 01, r_N_10 of the concrete model only attempts the west direction, and signals a routing failure if the attempt is not successful — r_N_10 does not redirect this packet back to arb_N_10, because it can infer that this packet has been diverted already, based on its location and the packet's destination. Assume a node only generates packets destined for nodes other than itself. So a packet destined for node 01 can only be generated by node 00, 11, or 10. The first choice to forward this packet is north for node 00, and west for node 11, as either node has two equal choices to send the packet and the first choice is to the shortest path to node 01. For node 10, the preferred choice for this packet is west. Note that none of

the preferred choices for node 10 go through r_N_10, which means that if r_N_10 receives a packet for node 10, it must have been diverted.

The router r_N_10 is modeled by the LNT process "*r_N_10_abs*" in Figure 4.10a. The first gate parameter "inp" corresponds to the incoming link (connected to the south arbiter of node 11). The next three gate parameters "out_PE", "out_W", and "out_N" correspond to the three externally visible communication links (r_N_10 → arb_PE_10), (r_N_10 → arb_W_10), and (r_N_10 → arb_N_10) in Figure 4.4, respectively. The fifth gate parameter " fail " does not correspond to a physical communication link but is used for making routing failures visible. The router process uses the variable "pkt" of type Bool to store the (contents of the) packet it forwards. Because the router r_N_10 can never make an illegal turn, the Boolean value contained in a packet is never consulted. Variable "arb_status" of type Bool stores the condition status of its output link connected to an arbiter, to which the router is attempting to forward the packet.

The behavior of process "*r_N_10_abs*" is a nonterminating loop consisting of a rendezvous "inp (?pk)", which synchronizes on gate "inp" and stores the received packet in variable "pkt", followed by a nondeterministic choice between sending the packet to its own PE (gate "out_PE", connected to the arbiter arb_PE_10), or forwarding the packet first to the west (gate "out_W", connected to arb_W_10), or to the north (gate "out_N", connected to arb_N_10). Two-way gate rendezvous is used to model packet forwarding: the gate on the router's side can synchronize with the gate on its connected arbiter's side (not shown on the figure). In the nondeterministic choice, the packet is sent only to a gate that is ready for synchronization; if more than one gate is ready, the choice is nondeterministic; if no gate is ready, the process waits until one of the gates becomes ready. During the rendezvous on a gate, both processes can exchange offers. For example, the "out_PE" gate represents the router's side of the communication link "r_N_10 → arb_PE_10"; hence the value of the packet "pkt" is passed to the receiving arbiter arb_PE_10. A rendezvous on a gate can only happen if both participating processes are ready; otherwise a gate blocks process execution when it waits for synchronization. In the second choice, r_N_10 checks the output link status of its connected arbiter arb_W_10 before sending a packet. This is represented by the rendezvous "out_W(?arb_status)", which waits to receive the status from this arbiter. It sends the packet west through "out_W" if the link is available, otherwise it tries to direct it to the north. If neither choice is feasible, the router performs a rendezvous on gate " fail "; this rendezvous is always possible, because it is not synchronized with any process. This gate rendezvous is referred to as "*route-failure* rendezvous" in the rest of this chapter. Execution

of a rendezvous on a gate produces a transition labeled with the gate's name and the values of the offer (if any). Notice that the order of the nested "**if**−**then**−**else**" constructs depends on the router, reflecting the asymmetry of the routing function.

The reason why the abstract model does not contain a route failure is that *r_N_10_abs* provides an alternative choice for *any* packet. Moreover, it does not use a packet's diversion status to limit the redundant alternative choice. Consider the case of a packet destined to another node (those destined to 10 can always be routed without failure). Because the other arbiter that r_N_10 connects to, namely arb_N_10, is not faulty in the situation depicted in Figure 4.9, this means that r_N_10 always sends the packet to arb_N_10 so that the "fail" gate rendezvous never occurs. To refine the abstract model, the router should only divert a packet to an alternative route if it has not been diverted already. Figure 4.10b shows the added check for the diversion status for the second choice in Figure 4.10a. A similar correction is made on the third choice, which is omitted on Figure 4.10b.

Although the general principle for the fault-tolerance routing is to provide as much adaptivity as possible, this error in the abstract model illustrates that making multiple

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

(a) The original LNT process for abstract r_N_10.

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

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

Figure 4.10: The original and modified LNT process for abstract r_N_10.

diversions can introduce incorrect functional behavior. As described above, r_N_10 can infer the diversion status from a concrete packet destined for node 01 and use it to avoid multiple diversions. On the other hand, if r_N_10 keeps diverting a packet destined for node 01 back to the direction it comes from, it is possible that this packet gets stuck in an infinite livelock loop: r_N_10 → arb_N_10 → r_S_11 → arb_S_11 → r_N_10, as shown in Figure 4.11. Livelock is a scenario where a packet circles around a loop infinitely often without ever reaching its destination. Therefore, avoiding multiple diversions on a NoC is an effective way to prevent livelock. On the fault-free two-by-two NoC in Figure 4.11, there are only two circular paths, the clockwise inner path and the counterclockwise outer path, that a packet can take to reach any of the four nodes. A router diverts a packet only if it is unable to continue forwarding it on the current circular path due to a link fault. Every time a router diverts a packet, it switches the packet from its current path to the other, effectively reversing its routing direction. The router hopes to deliver the packet through the alternative path. However, if the packet encounters another faulty link on the alternative path, making another diversion puts the packet back to its previous failure path, and the packet is guaranteed to hit the previous faulty link again before it reaches the destination. So having multiple diversions allows a packet to infinitely circle around a loop, which is formed by segments of the two circular paths, i.e., the connected routers and arbiters except for any processing element arbiter (arb_PE_xy).

4.3.2 Eliminating Livelock

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

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

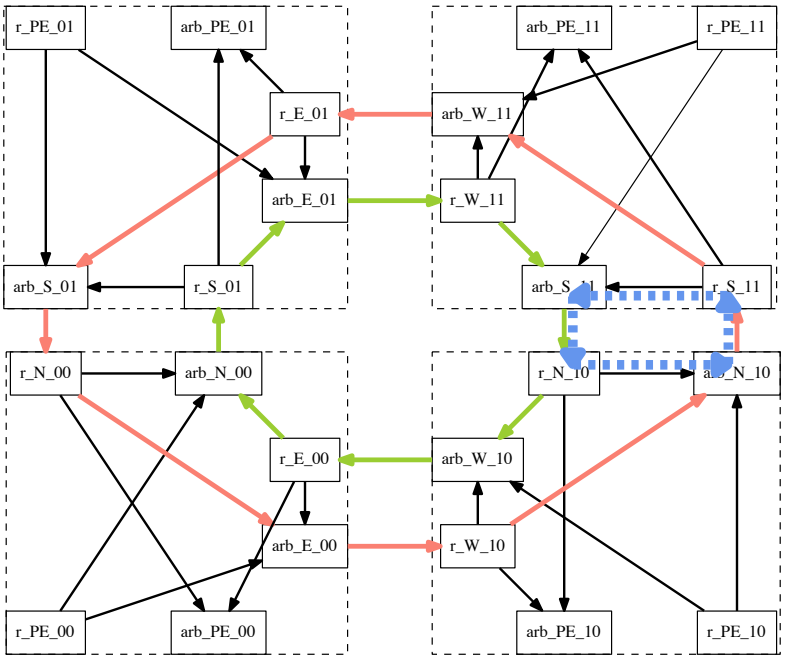


Figure 4.11: Illustration of the two circular paths that a packet can reach any node and the livelock loop r_N_10 → arb_N_10 → r_S_11 → arb_S_11 → r_N_10.

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

In general however, inferring the diversion status is difficult as a router does not know a packet's entire forwarding history. As an example, consider all packets that r_N_10 has to forward to its neighboring nodes. It receives all south-going packets from arb_S_11, and then sends them to either arb_W_10 or arb_N_10. Packets destined for node 11 (arb_PE_11) are not sent through arb_S_11 and hence do not reach r_N_10, and packets destined for node 10 (arb_PE_10) are not forwarded by r_N_10 to its neighbors. This means that packets of interest are destined for either node 01 (arb_PE_01) or node 00 (arb_PE_00). We know from the previous analysis that all packets destined for node 01 must have been diverted to reach this router. For packets destined for node 00, if they are

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

Figure 4.12: The new LNT process for r_N_10.

generated at either node 01 or node 10, then they must have been diverted before reaching r_N_10 as they failed on their preferred choices, i.e., their respective shortest paths to node 00. If a packet is generated at node 11, then its diversion status is not clear as it could be diverted if r_PE_11 forwards it west first or not diverted if south is the first choice. This uncertainty makes it impossible for r_N_10 to infer the diversion status for a packet, since it has no information about the packet's source and its routing preferences.

According to the original description [33] of the routing algorithm by Wu et al., there is no defined order between the two equal routing choices. This means that the implementation of the routing algorithm can bias towards one choice without violating the routing rules. For example, if r_PE_11 always chooses west over south for all packets destined for arb_PE_00, then the said uncertainty at r_N_10 can be resolved. This means that all packets received by r_N_10 are diverted, and this router does not need to divert them again by sending them back north, and the link from r_N_10 to arb_N_10 can be removed. The new LNT process for r_N_10 is shown in Figure 4.12. After receiving a packet on its "inp" gate that is connected to the output of arb_S_11, it extracts the packet's destination and stores it in "pkt_dest". It then compares the coordinates of the packet's destination with its own location — as usual, the "." notation expresses access to the field of a record. If the destination is reached, it delivers the packet by synchronizing on gate "out_PE" with arb_PE_10. Otherwise, it forwards the packet west if arb_W_10's output link is functional, and fails if it is faulty. A symmetric improvement is made to r_S_01 by tweaking r_PE_00 to make east as its preferred route. This resolves a similar uncertainty that all packets generated by r_PE_00 and destined for arb_PE_11 are routed to the east first. The result is that r_S_01 only receives diverted packets to forward to its

east, and the router does not need its output to arb_S_01. This breaks the livelock loop, i.e., arb_N_00→r_S_01→arb_S_01→r_N_00→arb_N_00. Moreover, this modification makes the north-to-south illegal turn disappear, preventing packet drop at this router. Note that the assigned ordering between two equal choices is only limited to the mentioned two PE routers when they forward packets destined for nodes in their diagonal directions. Both are equal choices for routing such a packet since each have the same distance to the destination, and therefore no performance penalty is introduced.

Since livelock always occurs on a closed path on the routing architecture, i.e., a path formed by alternating routers and arbiters that makes a packet circle around indefinitely, we may find livelock by identifying closed paths. For example, a packet destined for arb_PE_11 loops infinitely on the closed path if the output links of arb_N_00 and arb_N_10 are faulty: arb_E_00 → r_W_10 → arb_W_10 → r_E_00 → arb_E_00. In this case, r_E_00 can be modified to not divert the packet back east, but only send it north. The resultant NoC architecture is shown in Figure 4.13.

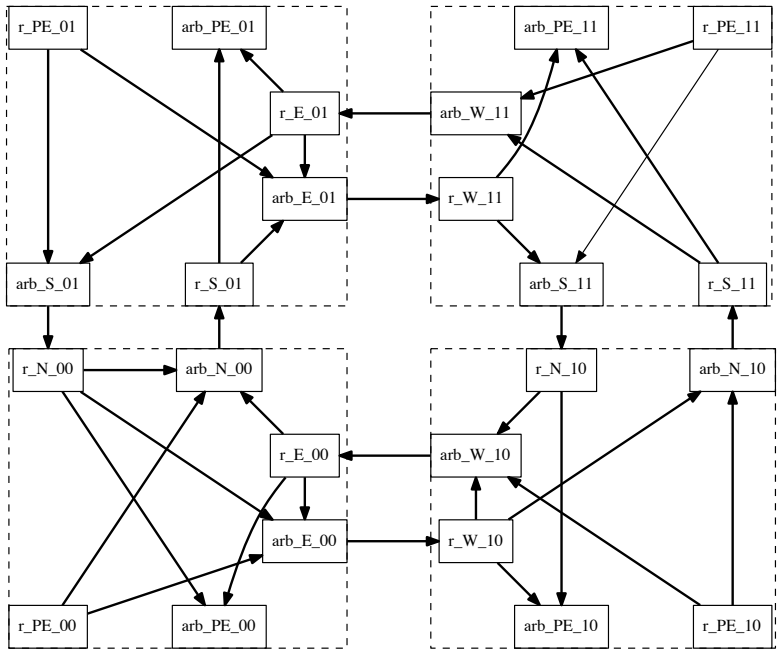


Figure 4.13: Improved two-by-two NoC architecture with livelock removal. Note that the modifications to avoid livelock have led to the removal of four connections that are no longer needed (i.e., between r_S_01 and arb_S_01, r_W_11 and arb_W_11, r_S_11 and arb_S_11, and r_N_10 and arb_N_10).

Note that multiple diversions still exist in some routers in Figure 4.13. This is because a packet's destination information alone is not sufficient to determine its diversion status at these routers. For example, the packets received by r_E_01 can be diverted if they are forwarded by r_S_11, and not diverted if they come from r_PE_11. However, livelock does not occur even if r_E_01 diverts a packet forwarded by r_S_11 again, because this packet is sent to r_N_10, where it gets squashed to avoid multiple diversions. While the improved routing algorithm cannot guarantee that a packet is not diverted more than once, it does guarantee that it is not diverted infinitely often for the two-by-two NoC. This property, however, needs to be formally verified.

Figure 4.14 presents the decision tree of the livelock-free routing protocol in pseudo code. For simplicity, this pseudo code only describes the decision, and it does not provide details on the communication between the routers and arbiters. In particular, determining if a link is faulty requires a communication with the arbiter to determine its current status. This communication is crucial to enable the protocol to adapt to transient failures. Furthermore, in the case of an illegal turn, the router must determine if the arbiter is busy, and in this case, if it is found to be busy, it drops the packet. Notice that also the modified routing protocol is not symmetric, because the possible routes depend on both the address of the router and the destination of the packet.

4.4 Verification Results

We applied a two-phase approach to verify a NoC model using the Cadp toolbox: first generating the Lts from the Lnt specification, and then analyzing the Lts to verify properties of interest. The Lts for each investigated model is generated compositionally [123], i.e., by first generating and minimizing the Ltss for each process separately before alternating combination, hiding, and minimization operations to obtain the Lts of the complete system. For the combination, we applied smart reduction [121], which uses heuristics to find an optimal ordering of the combination and minimization operations to keep the intermediate Ltss manageable. The maximal number of Ltss that can be composed in one step is set to 5. Hiding operations transform into internal transitions those labels that are no longer necessary for further synchronisations or the verification of the properties of interest, greatly enhancing the effectiveness of minimization and verification [124]. To ease the verification tasks, we hide all labels, except those corresponding to route failures and packet drops. Minimization is performed with respect to divergence-sensitive branching bisimulation equivalence [107].

A desktop machine with a CPU of eight 3.60 GHz cores and 16GB of available RAM is

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

Figure 4.14: Pseudo-code of the routing protocol.

used to generate the results in this section. One core is used at any time for the parallel composition and state minimization steps. One interesting observation is that with the removal of links and simplification of the routing algorithm presented in Section 4.3.2, it is possible to completely generate the LTSs of models with concrete packets. All results presented here are based on the two-by-two NoC models with concrete packets. The properties of interest are: (1) the routing algorithm is free of deadlocks and always tolerates a single link fault; (2) it guarantees packet delivery without network traffic; and (3) it is free of livelocks.

4.4.1 Deadlock Freedom and Single-link-fault Tolerance

In all experiments described in this subsection, each of the four PE routers repeatedly executes the following steps: it first generates a packet with a nondeterministically chosen destination (except to itself), and then sends the packet to its next forwarding direction, which is determined based on the packet's destination. Since each PE router is free to nondeterministically send a packet to any possible destination at any time, our verification generates *all* possible network traffic patterns. It is necessary to model all possible network traffic for the verification of deadlock freedom, since a deadlock can only occur when multiple packets in the network create a communication cycle as described in Section 3.5, and this cycle is broken by dropping one of the packets. Table 4.1 shows the LTS information for nine two-by-two mesh models: the first row represents a mesh without any link failure, and the remaining eight rows each represent the same NOC with one failure link whose location is shown in the first column. The columns under "Largest Intermediate LTS" show the number of states and transitions of the largest intermediate LTS, and the columns under "Final LTS" show those of the LTS corresponding to the complete model obtained at the end of the compositional generation. The two columns under "Performance" display the maximal amount of allocated virtual memory (in MB) and the total execution time (in seconds) to generate each LTS.

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

Table 4.1: LTSs of the two-by-two NOCs generated for the verification of deadlock freedom and one-fault tolerance.

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

To model a single link fault in LNT, a working arbiter process is replaced by an arbiter that sends false status to all its connected input routers.

Deadlock freedom is a global property, which requires reasoning about the complete system, and cannot always be inferred from the components taken in isolation. A system has a deadlock if its LTS contains a state/transition sequence that starts from the initial state and ends in a terminal state, i.e., a state without outgoing transitions. A straightforward approach for deadlock detection is to search for such states in the corresponding LTS. Using the CADP toolbox, it is found that no such sequence exists in any LTS of Table 4.1. Note that large intermediate LTS does not correspond proportionally to high memory usage. For example, the size of the largest intermediate LTS in the "00 → 01" model is larger than that in the "00 → 10" model, but it consumes less memory. The largest intermediate LTSs in the "01 → 11" and "10 → 00" models are substantially different in size, but they have comparable memory usage. All experiments show that the peak memory usage is a result of minimizing the largest intermediate LTS in each model. Memory usage for state minimization is, however, not only determined by the size of the LTS, but also determined by the branching structure of the LTS. This explains the weak correlation between the largest LTS and the peak memory usage. It is also worth noting that the differences in the state counts clearly show the asymmetry of the routing protocol. Since the entries in Table 4.1 cover all possible single-link-fault configurations, we can conclude that the link-fault-tolerant algorithm is free of deadlock for the improved routing algorithm on the two-by-two NoC.

To prove that a router is always able to route a packet, it is necessary to verify that no route-failure gate rendezvous occurs. Since these gates are not hidden during parallel composition, it is straightforward to check their existence in each LTS. Table 4.2 shows that no transitions are labeled with route-failure labels. This table also shows that with a single failure link in the network, packets may be dropped, namely when the attempt to make an illegal turn could potentially cause a deadlock. The packet drop labels in this table show the location where the drop happens together with the destination of the dropped packet. For example, "drop_Sr_11!Coordinates(0,1)" means a packet destined for node 01 is dropped by the southern router of node 11. The internal transition label is indicated by "i". Therefore, in a highly congested network, dropping packets is likely to happen. Note also that the occurrence of packet drop is more sensitive to certain fault locations than others. Faults in one of node 00's two incoming links from node 01 and 10 are responsible for the largest variety of packet drops. But faults in one of node 11's two outgoing links can be entirely tolerated by the routing algorithm without any packet loss.

Table 4.2: Labels of the LTS's corresponding to two-by-two NoCs generated for the verification of deadlock freedom and one-fault tolerance.

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

4.4.2 Packet Delivery

After the successful verification of deadlock freedom and single-link-fault tolerance, it is important to thoroughly check that each packet can reach its destination. The models of interest for this verification task have at most a single link fault. From the analysis in Section 4.3.1, it is known that with two (or more) faulty links, certain routing failures are unavoidable, therefore packet delivery is not guaranteed. Since the routing algorithm guarantees single-link-fault tolerance, it makes sense to check packet delivery on models with at most a single faulty link. Moreover, Table 4.2 shows that with network traffic, some packets may get dropped instead of reaching their destinations, with even a single link fault. However, packet drop only occurs to avoid deadlock, which requires the existence of network traffic. This means that packet delivery can only be checked on models without any additional network traffic. Therefore, while packet delivery can be verified using the CADP toolbox, in this simple case, it can be ensured by simply confirming that the network remains connected after removing a single failing link.

4.4.3 Livelock Freedom

In Section 4.3.2, a series of diagnostic examples is provided to eliminate livelock issues, which led to simplifications of the NoC architecture. This subsection provides formal analysis of livelock freedom on the simplified NoC. Since it requires at least two faulty links to cause a livelock, with eight external links, there is a total number of $\binom{8}{2} = 28$ different combinations of fault locations. The livelock freedom verification is divided into 28 individual tasks in which each one generates a LTS from a NoC model with a unique pair of link faults.

Similar to the packet delivery verification tasks, only one packet is allowed in the network.

A single packet in the network is sufficient for livelock detection, since only a link fault can trigger packet diversion and with multiple diversions a livelock can potentially occur. Traffic may cause a packet to be dropped, but it never causes a packet to be diverted. Therefore, network traffic does not contribute to the cause of livelock. The same configuration for PE routers from Section 4.4.2 can be used here. As mentioned before, the drawback of this configuration is the introduction of deadlock. Besides, with two link faults, it is unavoidable to have a routing failure that causes deadlock. So, it is certain that some deadlock state exists on the final LTS. However, this fact does not change the results of livelock freedom verification. Livelock is characterized as the existence of an infinite loop on a LTS containing only internal transitions. The goal of checking livelock freedom is to guarantee the absence of such a loop on the final LTS. Thus, the existence of a deadlock state is of no relevance for this verification goal. It is, however, necessary to perform state minimization with respect to divergence-sensitive branching bisimulation equivalence to preserve any actual livelock loop in the NOC model.

For each of the 28 verification tasks, there are 4 different experiments in which one node generates a single packet and then becomes inactive while the other three nodes remain inactive. Gate hiding for each experiment is applied to all communication gates in the model. This means that all previously visible gates are hidden, including the routing-failure gates and the packet's generation and consumption gates. Note that gate hiding has the potential danger of turning a cycle into a livelock loop, which causes a false negative result on the final LTS. A cycle differentiates from a livelock loop in that it is a loop with at least one visible transition label representing a communication with its environment. This factor, however, is eliminated in the proposed experiment setting, since with a single packet in the network, there does not exist meaningful cycles of transitions, such as continuous generation of packets. Hence it does not hamper the detection of real livelocks.

Livelock freedom is verified by checking a simple property requiring the presence of a livelock cycle, which can be expressed in the *Model Checking Language* (MCL) [125] by the following formula:

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

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

It is found that none of the 112 experiments satisfies the livelock property, which proves

livelock freedom on the improved routing architecture. From all experiments, the largest intermediate LTS has 112,176 states and 718,564 transitions, and the longest runtime is 175.08 seconds. Each final LTS has a single deadlock state and no transitions. For all experiments, the peak virtual memory used is 114 MB and the total time is about 5.28 hours.

4.5 Conclusion and Discussion

The construction and refinement of the two-by-two NoC model provided us with two valuable insights. The counterclockwise routing example reveals a packet leakage path in the arbiters that instructs their preceding routers to drop the packet. This leakage is due to the arbiter's model, in that each arbiter must check its succeeding router's availability before it can receive a packet from another router. For example, arb_W_11 must check with r_E_01 before it receives a packet from either r_PE_11 or r_S_11. Otherwise, if the arbiter does not receive its succeeding router's acknowledgement, it sends a "drop" signal to its preceding router. This option is modeled simply as the arbiter sending back the "drop" signal, which opens the path for packet leakage. The second lesson is that it is necessary for an arbiter to have a buffering capacity for the proposed routing architecture because an arbiter does not need to guarantee the availability of its succeeding router before it receives a packet. It is this idea that leads us to redesign the arbiters, such as the one shown in Figure 4.8. The state explosion problem encountered during the evolutions of our NoC models inspired us to implement a data abstraction scheme. This process provided us with a deeper understanding of the routing algorithm. With the data abstraction, routers can be categorized into RI0, RI1, and RI2, as described previously, and each category corresponds to one type of arbiter, as well.

To enable the verification of packet delivery, this chapter presents a hybrid modeling scheme as an extension to the previously presented pure abstract model. The discovery of routing failure hidden by the abstract model for router r_N_10 leads to the detection of the potential danger of multiple packet diversions in the original routing algorithm. It is found that excessive fault-tolerance in terms of multiple packet diversions not only fails to increase the chance of delivering a packet, but also potentially causes livelock problems where a packet circles around an infinite loop and never reaches its destination.

Multiple diversions are then analyzed in detail through a series of diagnostic examples on the concrete NoC model. The routing algorithm is corrected and certain routers are restricted to avoid multiple diversions and eliminate potential livelock loops. Since the

diversion status is not directly encoded in a concrete packet, it is not always possible to infer from the packet's destination information. Therefore, biased choice between two equally preferred routes is assigned to some PE routers, so that only diverted packets are received by those routers that cannot always decide upon the diversion status of packets. These modifications make it possible to remove redundant communication links on the routing architecture. This simplification eventually leads to manageable state space generation of the concrete model for the two-by-two NoC.

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

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

CHAPTER 5

FORMAL ANALYSIS USING LEMA

This chapter describes an alternative approach for modeling and verification using LPN and LEMA. Different from LNT, which uses rendezvous actions to represent progress of a system, LPN, defined in Section 2.3.4, uses global variables to encode a system's progress. Global variables can be used to enable or disable LPN transitions when they are assigned new values. State reachability analysis from CADP produces one or more LTSs whose transition labels store crucial information between states, and that from LEMA produces a state-transition graph where each state stores global and local variable values, place markings, etc. Our focus of state reduction techniques is also different: for CADP, it is on compositional minimization with respect to divergence-sensitive branching bisimulation, and for LEMA, the focus is on a POR technique based on the *ample set* method for LPNs.

POR techniques have proven to be successful at reducing the state space by ignoring unimportant concurrency in the model. The POR technique presented in this chapter analyzes transition dependencies through a recursive trace-back search on LPNs. A set with the smallest number of transitions that need interleaving is selected at each state. This approach preserves safety properties[1] and deadlock conditions by preserving traces that can potentially lead to their occurrences. This chapter then compares the cost and benefit of including trace-back in the ample set calculation in terms of state reduction, memory, and runtime for a series of buffers that uses asynchronous communication. These results are compared to those produced from a series of corresponding LNT models. VHDL models for representative routers and arbiters are described for the livelock-free link-fault-tolerant routing algorithm. They are constructed using a subset of the VHDL syntax that can be compiled to correct LPNs by the LEMA tool. Finally, observations are described on the verification results using POR with trace-back on components of the two-by-two NOC.

[1]We restrict to safety properties that can be encoded as a LPN failure transition.

5.1 Background and Related Work on POR

A major source of state explosion in many model checking tasks is the need for modeling concurrency by interleaving. For systems that are highly concurrent, such as most asynchronous circuits and protocols, techniques based on POR have proved to be highly effective at reducing the number of reachable states. *Commutativity*, also known as *independence*, is a common underlying principle for choosing a sufficient subset of transitions to fire in many sophisticated POR algorithms [25, 126–128]. POR techniques exploiting commutativity establish an equivalence relation based on all sequential executions of a system, and explore at least one representative execution from each equivalent class, while ignoring others in the same class. The basic idea is that instead of exploring all possible interleavings of concurrently enabled transitions, only a provably-sufficient subset of the enabled transitions at every state is selected, and only the states resulting from the firings of these transitions are explored.

Peled's *ample sets* method [20] selects the minimal number of enabled transitions that need interleaving at each state. *Stubborn sets* method [22, 23, 129, 130] maps statically determined dependencies between places and transitions of a Petri-net to a dependency graph, and it finds a *strongly connected component* (SCC) on the dependency graph containing at least one enabled transition at each state [3]. An optimal result for maximal reduction on deadlock-preserving strong stubborn sets [131] selects a SCC with the minimal number of enabled transitions. Godefroid's *persistent sets* method [24, 132, 133] builds a subset of enabled transitions that do not interact with and are not affected by transitions outside the persistent set in all states reachable from the initial state. The *sleep sets* method [24, 134] reduces concurrent transition firings but retains full state space. It saves significant effort for state matching [135] by reducing state revisits. A state search algorithm integrated with a coupling between sleep sets and persistent sets [25] or stubborn sets [136] allows one not to store an already visited state, as the probability of exploring a previously visited state again is very small. Algorithms for these reduction methods have been integrated in various on-the-fly model checkers (e.g. [23, 24, 137]). Peled and Godefroid's methods [21, 138] have been implemented as extensions to the SPIN model checker.

The key ideas behind these approaches are similar in that they all attempt to approximate the transition dependency, which is used in the state exploration to construct a sufficiently small subset at each state to prune unnecessary interleavings, while ensuring correctness of the verification result. The amount of analyses invested both statically and dynamically in the reduction can determine the precision of the approximations, and consequently affect the

size of the dependent transition set at each state [25, 131]. Unfortunately, it is not, in general, practical to construct a precise and yet minimal dependent set. Valmari proves in [131] that detecting deadlock for the state reduction on a *1-safe* Petri-net is PSPACE-hard. The accuracy in the dependency prediction always come with high computational complexity. Theorem 11 in Chapter 10 of [2] proves that checking the dependency condition for a state and its subset of enabled transitions is at least as hard as checking the reachability of the full state space. It is therefore important that this approximation retain a good balance between a too coarse approximation and a too fine one, as the former renders reduction which does not solve the state explosion problem and the latter is too expensive to compute. The quality of dependency relation analysis can significantly affect the results of state reduction. In recent years, a *dynamic* POR (DPOR) approach for software model checking has been proposed by Flanagan et al. [126] to dynamically track interactions between concurrent processes/threads, and this information is used to refine the construction of persistent sets at run time. Rodríguez et al. [139] proposes an optimal DPOR algorithm that leverages cutoff events in *net-unfolding* to effectively prune the number of explored Mazurkiewicz traces [140], and uses state caching to speed up event revisits.

5.2 POR with LPNs

In Section 2.3.4, detailed examples are given to show how the enabling and disabling of LPN transitions can be used to check certain safety properties. To summarize, the three types of failure behaviors that can be checked in our LPN models are

1. Enabling of a failure transition;
2. Disabling of a disabling failure transition; and
3. Deadlock (i.e., no transition is enabled in some state).

A correct verification algorithm must ensure that none of these failures is missed. The critical step to partial order reduction is the construction of the ample set (i.e., the subset of the enabled transitions which must be interleaved). A smaller ample set leads to more state reduction, but care must be taken to ensure that a sufficient set of transitions are included such that none of the stated failure behaviors are missed. The correctness of this approach is discussed in Section 5.2.4.

To construct the ample set, it is necessary to understand the dependency between transitions, and that no dependent transitions are missed in any cycle in the state space. This section describes our method of constructing an ample set that leverages trace-back

to reduce their size. The top-level algorithm is shown in Algorithm 5.1. At state s, each enabled transition t is used as a seed to construct a possible ample set. For each enabled transition t in state s, there exists a *dependent* set, which is a set of transitions consisting of t itself and transitions that must interleave with t due to dependencies. The algorithms for computing this dependent set are adapted from the ones presented in [141] and [142] with modifications as needed for our LPN model and correctness conditions.

The computed dependent set is put in the `ampleQueue`(s). The `ampleQueue`(s) is a priority queue: the dependent set of the minimal size is put on top of the queue; if two sets are of the same size, the set constructed by the seed that had less chances to fire before is put on the front of the queue. The top of the queue is returned as the ample set at state s. Organizing the queue in this way helps impose fairness on transition firings. It may potentially reduce the burden of firing concurrent transitions when a cycle needs to close during the state space search. If a transition is enabled at each state of a cycle of states, it has to get a chance to fire in at least one of those states in the cycle. The strong cycle condition [2, 143] requires that one state in the cycle fully expands all interleaving transitions to avoid missing the firing of any consistently enabled transitions in the cycle. This mechanism, however, may degrade the result of the state space reduction by forcing independent transitions to interleave. Our improved cycle closing condition adapts an analysis technique proposed by Zhang et al. [144]: it expands only a set of necessary interleaving transitions. With the imposed fairness in transition firings in `ampleQueue`(s), this set can get even smaller.

5.2.1 Preparations

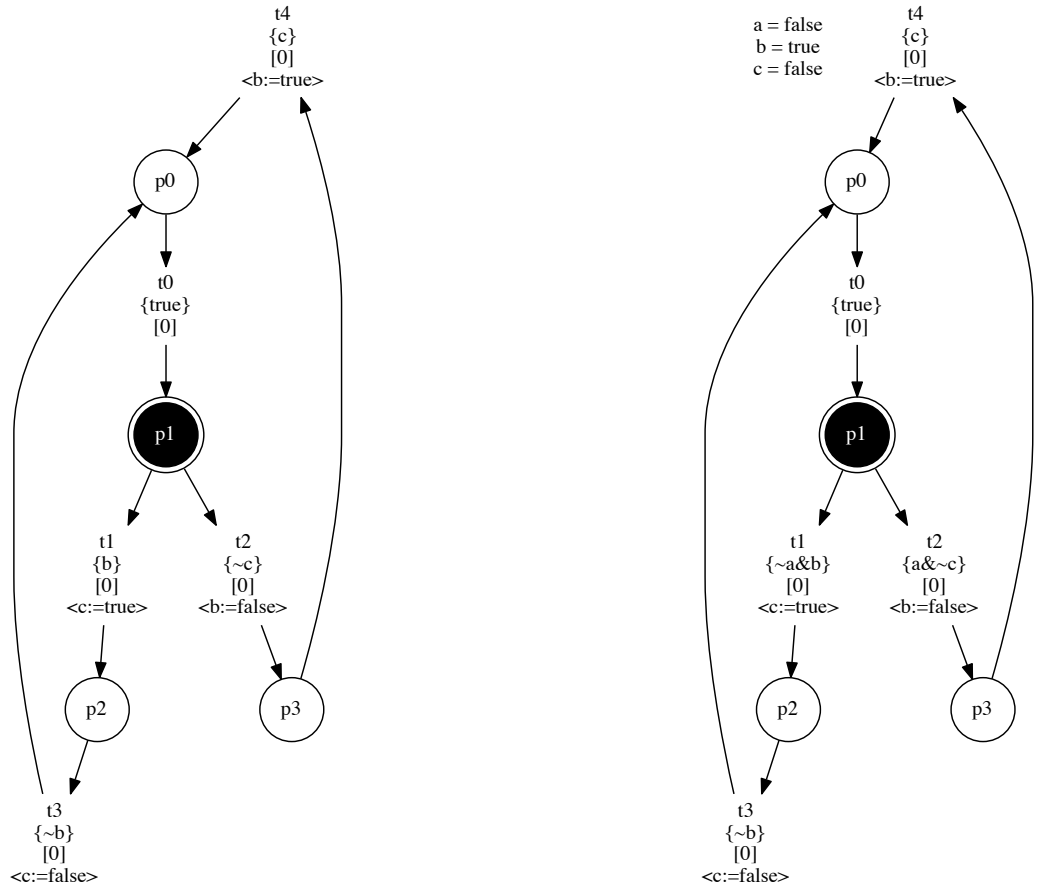
To improve the efficiency of the ample set construction, static analysis is performed before state exploration to derive structural information about the LPNs. First, the set `conflict`(t) is computed that includes all transitions that share preset places with transition t, as the firing of one transition in this set disables all other transitions in the set. Each transition t' in the `DisabledBy`(t) set satisfies one of the following conditions: t' is in the set `conflict`(t); the firing of t can potentially set the enabling condition for t' to **false**; or at least one of t's and one of t''s nonvacuous assignments assign to the same variable. The set `Disable`(t)

Algorithm 5.1 Ample set `ample`(s) computation.

```
for all t ∈ enabled(s) do
    π := dependent(s, t, ∅)
    enque(ampleQueue(s), π)
return top of ampleQueue(s)
```

is a collection of all transitions that can potentially disable transition t. This includes any transition t' that satisfies the following condition: t' is in the set $\texttt{conflict}(t)$ or $t \notin T_p$, and the firing of t' can potentially set t's enabling condition to **false**. The set $\texttt{canEnable}(t)$ is a set of transitions that can enable t by setting its enabling condition to **true**.

As described later, the size of $\texttt{DisabledBy}(t)$, $\texttt{Disable}(t)$ and $\texttt{canEnable}(t)$ determines the amount of analysis needed to perform trace-back. Refinements on any of these sets can potentially reduce unnecessary effort on LPNs' traversal. Let us consider two types of static refinements on $\texttt{DisabledBy}(t)$ and $\texttt{Disable}(t)$. More specifically, after these two sets are computed for each transition t according to the said conditions, the following screening steps are taken on every transition in these sets. First, if t belongs to a LPN process that does not have concurrency, then for any transition t' in the same LPN process, $t' \notin \texttt{DisabledBy}(t)$ and $t' \notin \texttt{Disable}(t)$, unless $t' \in \texttt{conflict}(t)$. In other words, for transitions in the same LPN process as t that does not allow concurrency, only transitions that are in conflict with t are kept in $\texttt{DisabledBy}(t)$ and $\texttt{Disable}(t)$. This refinement is based on the fact that nonconflicting transitions t and t' in a nonconcurrent LPN process are never simultaneously enabled as they never obtain their respective preset markings at the same time. In Figure 5.1a, $t4$ is included in $\texttt{DisabledBy}(t3)$ since its enabling condition can be set to **false** by $t3$. However, they can never obtain their preset tokens at the same time because no concurrency exists, and therefore $t4 \notin \texttt{DisabledBy}(t3)$ and $t3 \notin \texttt{Disable}(t4)$. Similarly, $t3 \notin \texttt{DisabledBy}(t4)$ and $t4 \notin \texttt{Disable}(t3)$. The second refinement prunes transitions with contradictory enabling conditions. If there are no variable assignments that satisfy the enabling conditions of both t and t', then they are never enabled simultaneously. If t' is in either $\texttt{DisabledBy}(t)$ or $\texttt{Disable}(t)$, or both, then it can be safely removed from these sets. For example, Figure 5.1b shows that $t1$ and $t2$ are in conflict as they share preset place $p1$. However, since they have contradictory enabling conditions, they can never be enabled at the same time. As for the algorithmic implementation, the first refinement requires identifying nonconcurrent LPN processes, where every transition t has exactly one preset place and one postset place (i.e., $|\bullet t| = |t \bullet| = 1$). The second refinement needs the implementation of a subset of the functionality of a standard *SAT solver* (i.e., the "unsatisfiable" part). Also, note that these refinements operate on the previously computed $\texttt{DisabledBy}(t)$ and $\texttt{Disable}(t)$ sets, and hence do not necessarily examine all transitions in a LPN model. These refinements are new, relative to the method described in [141, 142].



(a) LPN process with no concurrency.

(b) Contradictory enabling conditions.

Figure 5.1: Examples of refinements.

5.2.2 Dependent Set

The dependent set function, shown in Algorithm 5.2, finds all transitions that must interleave with the enabled transition t in state s. This set always includes transition t (line 1). Then, the algorithm checks transitions that can be disabled by firing t or can disable the firing of t (line 4). For each transition t_i in $\texttt{DisabledBy}(t) \cup \texttt{Disable}(t)$, if it is currently enabled and is not already included in the dependent set d, and either t_i is not a persistent transition or it can disable t (line 5), [2] then the algorithm recursively determines the dependent set for t_i and adds it to the current dependent set d (line 6). On the other

[2]We only need to check whether the persistent transition $t_i \in \texttt{Disable}(t)$ when it is enabled. Since the only way that t_i is disabled by t is if $t_i \in \texttt{conflict}(t)$, checking $t_i \in \texttt{Disable}(t)$ is sufficient as $\texttt{conflict}(t) \subseteq \texttt{Disable}(t)$.

Algorithm 5.2 Dependent set $\texttt{dependent}(s, t, d)$ computation.

```
1:  d := {t}
2:  if |d| = |enabled(s)| then                                  ▷ d has all enabled transitions.
3:      return d
4:  for all  t_i ∈ (DisabledBy(t) ∪ Disable(t))  do
5:      if (t_i ∈ enabled(s) ∧ (t_i ∉ d)  ∧  (t_i ∉ T_p ∨ t_i ∈ Disable(t))) then
6:          d := d ∪ dependent(s, t_i, d)
7:      else if  (t_i ∉ enabled(s)) then
8:          if necessary(s, t_i, d) ≠ null then
9:              for all t_j ∈ (necessary(s, t_i, d) − d) do
10:                 d := d ∪ dependent(s, t_j, d)
11:         else
12:             return enabled(s)
13: return d
```

hand, if t_i is not enabled, then the algorithm attempts to find the necessary set (described in Section 5.2.3) that can lead to the enabling of t_i. If the necessary set of t_i can be found (line 8), then for each transition t_j in the necessary set but not already in d, the algorithm finds its dependent set and adds it to the dependent set (lines 9-10). On the other hand, when no necessary set for t_i can be found, the enabled set at state s is returned (line 12). Finally, after all transitions have been considered, the algorithm returns the dependent set d (line 13).

5.2.3 Necessary Set

The purpose of constructing a necessary set for transition t_i in state s is to not only construct an ample set with the minimal size, but also guarantee that the resulting ample set does not miss the enabling of any potentially dependent transition that has a chance to become enabled. The idea is to start from the disabled transition t_i, trace back on one or more LPN processes to find enabled transition(s) that can help to enable t_i. The necessary set computation recursively calls itself to find two sets of transitions that can contribute to the firing of t_i: one that can help to bring the token(s) to the preset of t_i and one that can help to set the enabling condition of t_i to **true**. The one with the smaller size is selected to be the necessary set for t_i.

The algorithm for computing the necessary set is shown in Algorithm 5.3. To improve the efficiency of the necessary set calculation, a cache is maintained of previous results. The table $\texttt{cache}(s, t)$ stores previously computed necessary sets for transition t in state s. Therefore, the first step of the algorithm is to determine if the result already exists in the cache, and if so, return that result (line 1). Next, the algorithm creates a set n_m which includes enabled transitions that may help to get a preset place of t_i marked (lines 2-14). To compute n_m, each unmarked preset place, p_j, is considered. The algorithm checks its preset transition t_k.

Algorithm 5.3 Necessary set $\texttt{necessary}(s, t_i, d)$ computation.

```
1: if cache(s, t_i) is defined then return cache(s, t_i)
2: n_m := null
3: if •t_i ⊈ markedPlaces(s) then
4:     for all (p_j ∈ •t_i) ∧ (p_j ∉ markedPlaces(s)) do
5:         n_t := ∅
6:         for all t_k ∈ •p_j do
7:             if t_k ∈ enabled(s) then n_t := n_t ∪ {t_k}
8:             else
9:                 if t_k ∈ visited(s, t_i) then
10:                    if cache(s, t_k) is defined then n_t := n_t ∪ cache(s, t_k)
11:                else
12:                    visited(s, t_i).Add(t_k)
13:                    n_t := n_t ∪ necessary(s, t_k, d)
14:        if n_t ≠ ∅ ∧ ((n_m = null) ∨ (|n_t − d| < |n_m − d|)) then n_m := n_t
15: if n_m ≠ null ∧ |n_m| = 1 ∧ |n_m − d| = 0 then
16:     cache(s, t_i) := n_m
17:     return n_m
18: n_e := null
19: if eval(En(t_i), s) = false) then
20:     for all conjunct_j ∈ En(t_i) do
21:         if eval((conjunct_j, s) = false) then
22:             n_c := ∅
23:             for all t_k ∈ canEnConj(j, t_i) do
24:                 if t_k ∈ enabled(s) then n_c := n_c ∪ {t_k}
25:                 else
26:                     if t_k ∈ visited(s, t_i) then
27:                         if cache(s, t_i) is defined then n_c := n_c ∪ cache(s, t_i)
28:                     else
29:                         visited(s, t_i).Add(t_k)
30:                         n_c := n_c ∪ necessary(s, t_k, d)
31:             if n_c ≠ ∅ ∧ ((n_e = null) ∨ (|n_c − d| < |n_e − d|)) then n_e := n_c
32: if (n_e = null ∨ n_m = null) then n := getSmallerSet(n_m, n_e)
33: else n := getSmallerSet(n_m − d, n_e − d)
34: if n ≠ null then cache(s, t_i) := n
35: return n
```

If t_k is an enabled transition, it is then included in the temporary set n_t. If it is not enabled but has been visited by t_i before (line 9), the algorithm returns its stored necessary sets if it is in the cache. The hash table $\texttt{visited}(s, t)$ stores all the visited transitions by t at state s. If t_k is not stored in the cache, it means that a circular trace-back has been formed in that t_k was visited before by t_i but its necessary set has not been found. In this case, the algorithm skips searching for t_k's necessary set and considers the next candidate for $\bullet p_j$ (line 6). As we can see, it is possible to get into a circular search for necessary transitions for t_i and the use of a hash table $\texttt{visited}(s, t)$ breaks the cycle and guarantees termination of the necessary computation. If t_k has not been visited before (line 11), the algorithm adds it to the set of visited transitions for t_i and recursively computes the necessary set for t_k, and the recursion terminates when an enabled transition is encountered. For each unmarked

place p_j, the set n_m is updated if a nonempty necessary set n_t of a smaller size is found (lines 14). Lines 15 to 17 describe an optimization step in that a minimal necessary set has been found and the search for necessary set can terminate without further exploration. This condition (line 15) is satisfied when only one transition can bring a marking to a preset place for t_i and this transition is already included in the dependent set d. It is a locally optimal choice because no additional enabled transitions other than those in d are needed for the enabling of t_i. Next, the algorithm creates a set n_e which includes enabled transitions that may help enable t_i by setting the enabling condition to **true** (lines 18-31), provided that t_i's enabling condition evaluates to **false** at state s (line 19). If the enabling condition of t_i is in a conjunctive form, then each conjunct is considered separately, otherwise the entire enabling condition is considered as one single conjunct (line 20). For each transition t_k in `canEnConj`(j, t_i) (a set of transitions that can help enable the jth conjunct of t_i's enabling condition and $0 < j \leq$ `number_of_conjuncts`$(En(t_i))$), the algorithm recursively finds its necessary set and updates n_e until an enabled transition is encountered. The search for necessary transitions that can help enable t_i's enabling condition is actually the search for transitions that can help enable one conjunct of t_i's enabling condition. Although all conjuncts of t_i's enabling have to be **true** in order for it to fire, having necessary transitions that can help the enabling of one conjunct is sufficient in state s. If more conjuncts need to be enabled, then necessary transitions in future states will be searched until all conjuncts are enabled. It is therefore not necessary to interleave all transitions (by including them in the necessary set) that can help enable all conjuncts in state s. Finally, the smaller of n_m and n_e is selected as the necessary set, assuming a `null` set has the size of infinity (line 32 to 35). The resulting necessary set is stored in the cache and returned.

To illustrate the trace-back procedure, consider the ample set construction for the producer-consumer model in Figure 5.2. Note that this model is the same model shown in Figure 2.24 in Chapter 2, except that places, transitions, and some variables have been renamed to ease the explanation here. First, for the enabled transition $t0$, one of its assignments assigns s to **true**, which disables $t5$. However, $t5$ is not enabled in the initial state, therefore a necessary set must be calculated for $t5$. Transition $t4$ is necessary to bring the token to $p5$, but transition $t4$ is not enabled because $p4$ is not marked and s is **false**. To get a token to $p4$ requires transition $t3$, which is enabled. Setting s to **true** requires transition $t0$. The algorithm needs to only include one of these transitions, and it includes $t0$ since it is already included in the dependent set. Therefore, the dependent set for $t0$ only includes $t0$ itself, which means the ample set can include just this single transition, and it does not need to be

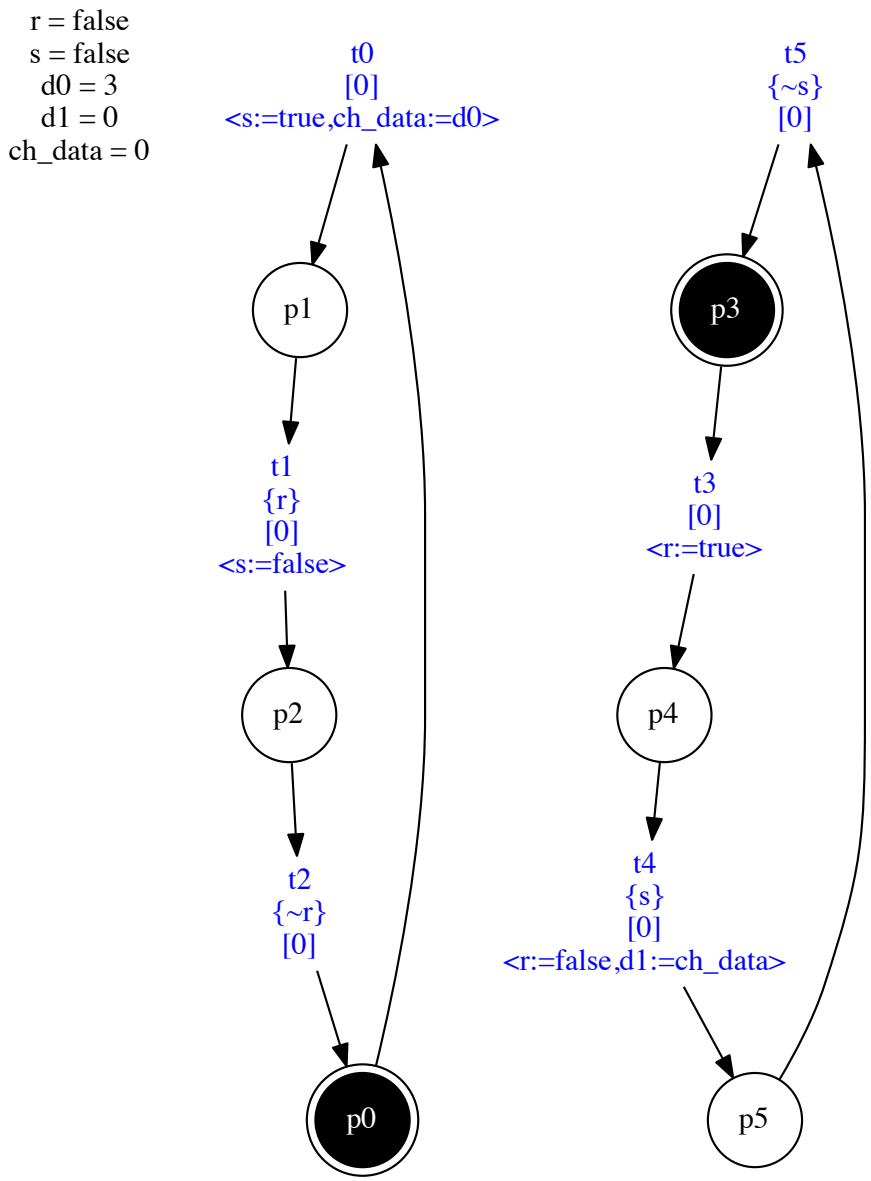
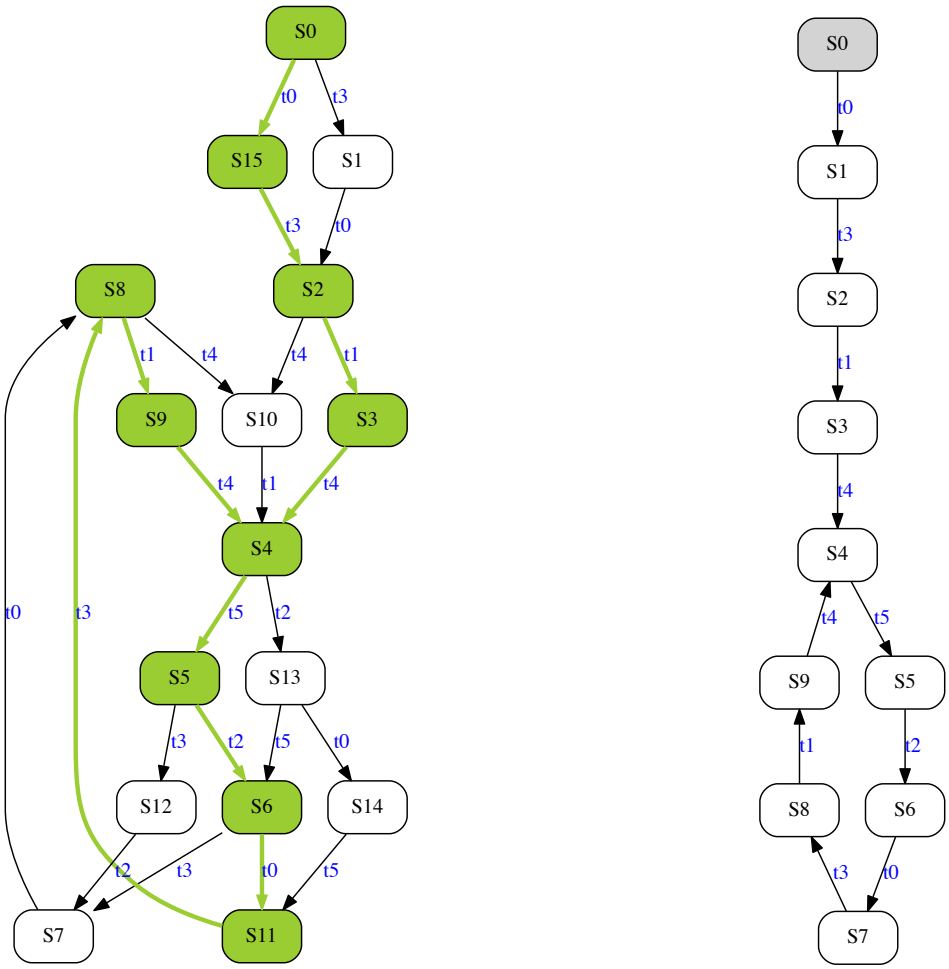


Figure 5.2: A simple producer-consumer LPN model.

interleaved with $t3$. State graphs from full state search and POR are shown in Figure 5.3. Transitions and states marked in green in Figure 5.3a are those selected by POR. As we can see, POR ignores unnecessary interleavings of transition firings by selecting and executing a representative sequence of transitions to fire. The resultant reduced state graph is shown in Figure 5.3b.

5.2.4 Correctness and Time Complexity

When the ample set construction is used to check the failure behaviors discussed in the beginning of Section 5.2, it is guaranteed to find a failing behavior when the LPN includes



(a) Full state graph.

(b) Reduced state graph

Figure 5.3: Full and reduced state graphs for the producer-consumer LPN model. Note that labels of each state is not showing here.

failures. For this to be true, it must be the case that it never misses the enabling of a failure transition. It must also not miss the disabling of a disabling failure transition. Preserving the disabling of transitions also helps to identify deadlock in the system. A system is said to deadlock if, at some state, no progress can be made. For LPNs, deadlock occurs when no transition is enabled. Such a behavior can be inherent in all paths, and is easily checked. However, it may be that a particular interleaving leads to the disabling of all possible enabled transitions. In this case, correctly preserving the disabling of transitions guarantees that the system deadlocks if there exists such a possibility.

The key element of the ample set construction is the computation of the dependency

relation. The dependency relation is defined by taking the set of complement of independent transitions (e.g., [25], [2]). A pair of independent transitions should not enable or disable each other, and commuting both enabled transitions should lead to a unique state. The dependency condition requires that a transition depending on any transitions in the ample set cannot occur before some transition from the ample set occurs first. This implies that such a dependent transition should be included in the ample set [143]. Given the properties that are checked on our LPNs, our dependency relation must be constructed in such a way that any transition that can be enabled gets a chance to be enabled and any transition that can be disabled gets a chance to be disabled. The dependent set is constructed using Algorithm 5.2. This algorithm checks the dependent transitions of a seed transition t in two sets: a set of transitions that can be disabled by t (i.e., `DisabledBy`(t)) and a set of transitions that can disable t (i.e., `Disable`(t)). For every transition t_i in `DisabledBy`(t), if it is enabled, is included in the dependent set of t. Next, the algorithm searches for any other transitions depending on t_i and includes those transitions in the dependent set of t as well. The iterative process terminates when all dependent transitions of t are found. On the other hand, if t_i is not enabled, the algorithm searches for the set of necessary transitions of t_i to be included in the dependent set of t. The idea of tracing-back to find necessary transitions complies with the dependency condition, and the reasoning is as follows. For every transition t_i depending on t, if t_i is not enabled, some other transition, say t_j, that leads to the enabling of t_i, should depend on t. Therefore, t must interleave with t_j and the chance of enabling t_i is preserved. A similar argument can be made for transitions in `Disable`(t). This case is used to preserve the chance of a transition to be disabled. Namely, any enabled transition t_i that can disable t must be included in the ample set. Also, if t_i disables t but is not currently enabled, trace-back is used to find, if one exists, another enabled transition t_j that contributes to the eventual enabling of t_i. In this case, the potential for disabling t is preserved. Therefore, the ample set construction always preserves the potential for any transition to become enabled or disabled. This result coupled with cycle closing, which, as described earlier, is used to ensure that a consistently enabled transition is not ignored. A verification method using this ample set construction is therefore correct. The correctness argument made here coincides with the criteria for the stubborn set construction on Petri-nets [131], whose correctness is formally proven.

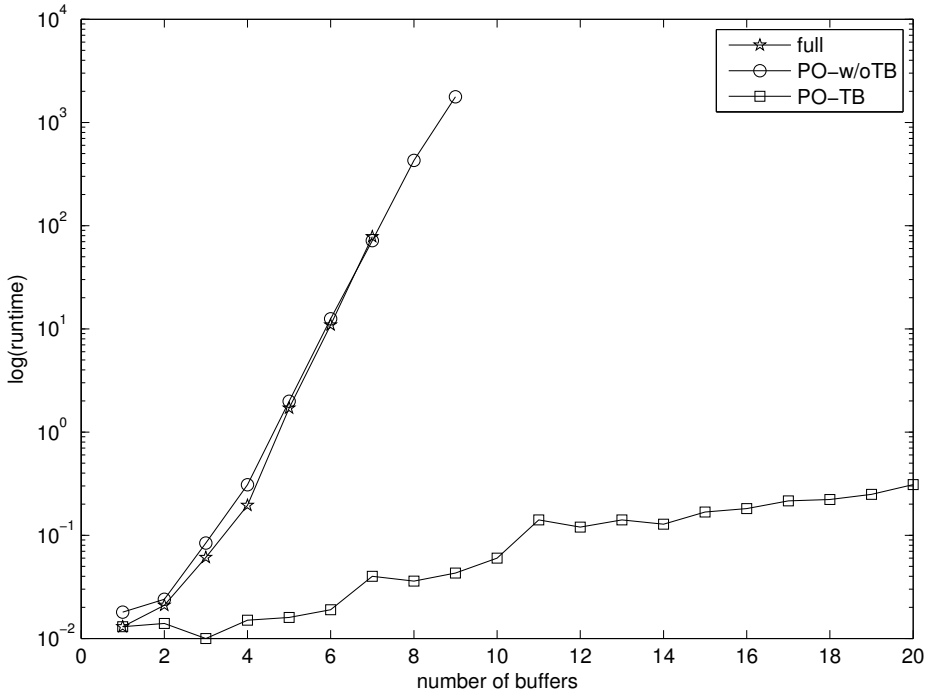
The time complexity is estimated by analyzing algorithms in the following order: Algorithm 5.3, 5.2, and 5.1. The necessary computation effectively walks on transitions of a LPN model, until an enabled transition is encountered. Since the necessary set for each visited

transition is cached, and circular search is prevented, the complexity is bounded by $O(n)$, where n is the number of transitions in the LPN. For the dependent computation of an enabled transition t, consider two extreme cases, assuming that t is dependent on almost all transitions. The first case is that almost no transitions are enabled in a state. This means that each dependent transition calls the necessary computation once. Since the complexity of necessary computation is $O(n)$, having n dependent transitions calling it results in a complexity of $O(n^2)$. The second case is that almost all transitions are enabled in the LPN, eliminating the need for necessary computation. Again, assuming that t is dependent on all transitions in the LPN, then each transition is added to t's dependent set, resulting in a complexity of $O(n)$. As for the ample set algorithm (Algorithm 5.1), the first case implies that in its loop over enabled transitions, there is a negligible number of calls to the dependent function, and the complexity is $O(n^2)$, as it is dominated by the necessary set computation. In the second case, the complexity of the loop over enabled transitions is $O(n)$, as almost all transitions are enabled, and each iteration calls the dependent computation once, whose complexity is $O(n)$. Therefore, the complexity in the second case is bounded by $O(n^2)$. Overall, in both cases, the time complexity for the presented ample set algorithm is $O(n^2)$.

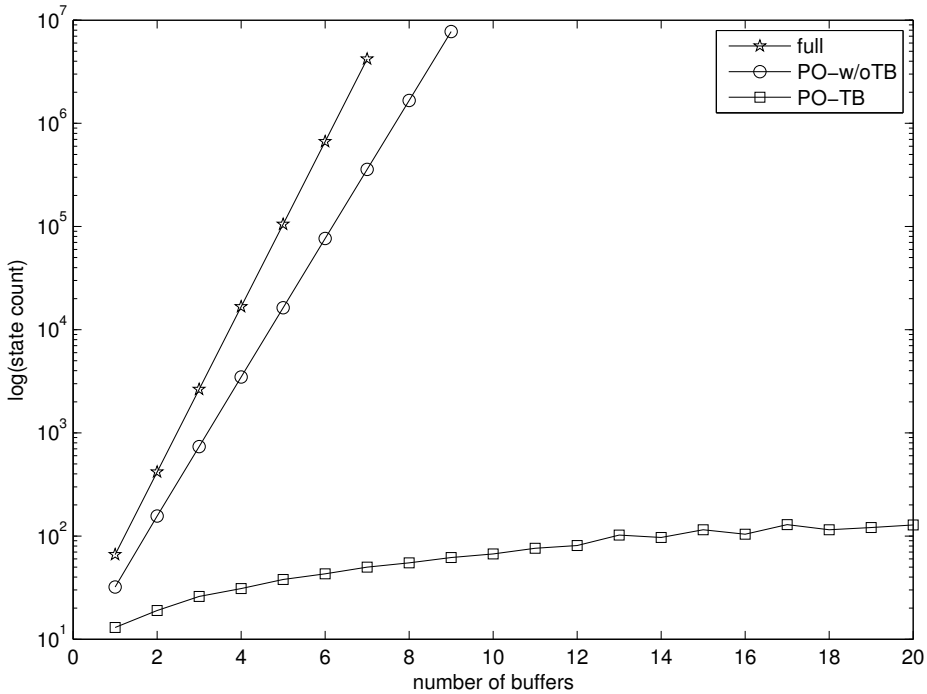
5.2.5 Evaluation of Trace-Back

The proposed partial order reduction technique on LPNs is implemented in Java and performs state reductions on a standard depth-first search algorithm. Experiments have been performed on a series of buffers that use asynchronous communication and several nontrivial asynchronous circuit designs. All experiments are performed on a Linux machine with 4GB memory and a quad-core 2.8GHz processor. The time limit for running each test case is set to 4 hours.

To evaluate the cost and benefit of trace-back, the LPN model in Figure 2.24 is extended by inserting a series of buffers between the producer and the consumer. Communication between any two connected modules (i.e., producer, consumer, buffers) is asynchronous, and the model in Figure 2.24 depicts the communication protocol. Concurrency increases significantly as the number of buffers increases and the resulting state space grows exponentially. Experiments are performed on 20 models, each with an increasing number of buffers from 1 to 20. Figure 5.4 shows comparisons of their runtimes and state counts. The runtime for state exploration increases significantly when trace-back is turned off, removing most of the benefit of partial order reduction, but significant improvements are possible with a partial order method that uses trace-back. Indeed, the state count grows in a nearly linear fashion, enabling much larger models to be analyzed.



(a) Runtime.



(b) State count.

Figure 5.4: Runtime and state count for the buffer examples with 1 to 20 buffers.

Table 5.1: Results for several asynchronous circuits models.

Designs	Dfs			Spin-Por			Por-Tb		
Name	Time	Mem	$\|S\|$	Time	Mem	$\|S\|$	Time	Mem	$\|S\|$
arbN3	0.357	17.4	3756	0.860	0.4	3756	0.644	33.5	397
arbN5	3.431	363.0	227472	1.44	79.8	227472	2.031	208.6	2466
arbN7	m-out	m-out	m-out	261	3196.8	6758278	6.119	208.9	8980
arbN9	m-out	m-out	m-out	m-out	m-out	m-out	49.5	333.2	58814
arbN11	m-out	m-out	m-out	m-out	m-out	m-out	462.952	420.5	493369
fifoN3	0.146	8.4	644	0	0.1	644	0.172	11.1	29
fifoN5	0.493	37.3	20276	0.07	4.5	20276	0.266	16.2	77
fifoN8	63.891	919.3	3572036	26.5	1144.6	3572036	0.956	33.8	882
fifoN10	m-out	m-out	m-out	m-out	m-out	m-out	2.039	130.9	2286
dmeN3	4.924	356.3	267999	0.18	15.2	117270	0.991	66.2	402
dmeN4	m-out	m-out	m-out	11.2	785.3	4678742	2.358	208.8	1665
dmeN5	m-out	m-out	m-out	m-out	m-out	m-out	44.482	209.0	29919

The second analysis involves the state space search for several asynchronous circuit designs. They include a tree arbiter (Arb) of multiple cells [145], a self-timed first-in-first-out (Fifo) design [146], and a distributed mutual exclusion element (Dme) with a ring of Dme cells [145]. The results are shown in Table 5.1. The results in columns under Dfs are from traditional depth-first search without any state reduction. The columns under Spin-Por show results from the Spin model checker with its own partial order reduction [21], and those under Por-Tb show results of depth-first search with partial order reduction using trace-back. For each of these methods, the total runtime (Time), total memory usage (Mem), and state count ($|S|$) are compared. Also, an entry "m-out" in the table means the memory upper limit is reached.

From Table 5.1, it can be seen that the depth-first search (Dfs) method quickly fails on designs with a large state space, due to the state explosion problem. The results from Spin with partial order reduction (Spin-Por) show some advantages over the Dfs method in that more designs are able to complete. Spin exceeds the memory limit on arbN9, arbN11, fifoN10, and dmeN5. Partial order reduction with trace-back (Por-Tb) has effective reduction in the state count for all the designs listed in this table, compared to Dfs and Spin-Por, especially for designs with large size. The savings in state count leads to significant memory savings for the Por-Tb method. For the set of arbiters and Fifo designs, Spin-Por does not reduce the state count at all. It is possible that Spin finds the state space for arbN7 due to its efficient state storing mechanisms, such as *reliable hashing* [147] and *state-space caching* [148] and its improvement with the use of sleep sets [135], and therefore could

complete more designs than the DFS method. Also, note that SPIN is implemented in C whereas our DFS and POR are implemented in Java, which has a well-known memory overhead compared to a C implementation. The only set of designs showing reductions from SPIN-POR is the set of DMEs. The POR-TB approach, however, produces even smaller state count and consequently leads to memory savings as the size of the DME becomes larger. One drawback of using the POR-TB approach is the long runtime on large models such as arbN11 and dmeN5. This is because these circuit models contain many dependent transitions, which leads to extensive graph analysis on all LPNs for trace-back during state exploration. Overall, these circuit models indicate that POR-TB outperforms DFS and SPIN-POR for large designs by constructing a smaller state space with some runtime overhead.

A partial order reduction approach with behavioral analysis has recently been presented by Zhang et al. in [144]. This method generates full local state graphs first for each module of a LPN model. By analyzing the local state graphs which show state-transition relations, it determines the dependency relation for each transition and applies reduction when constructing the global state space. Figure 5.5 shows a comparison of state count between POR with trace-back and behavioral analysis. Note that the state count in this figure uses a logarithmic scale. The mixed results indicate that each method has advantages on different models, or the same set of models but with different scales. The runtime and memory results of the behavioral analysis reported on these examples show only a small amount of performance overhead occurred.

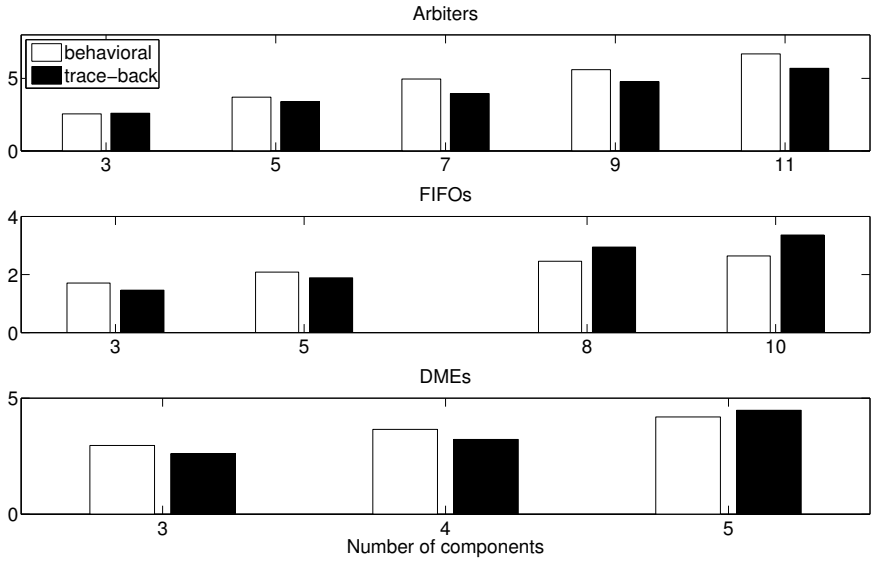
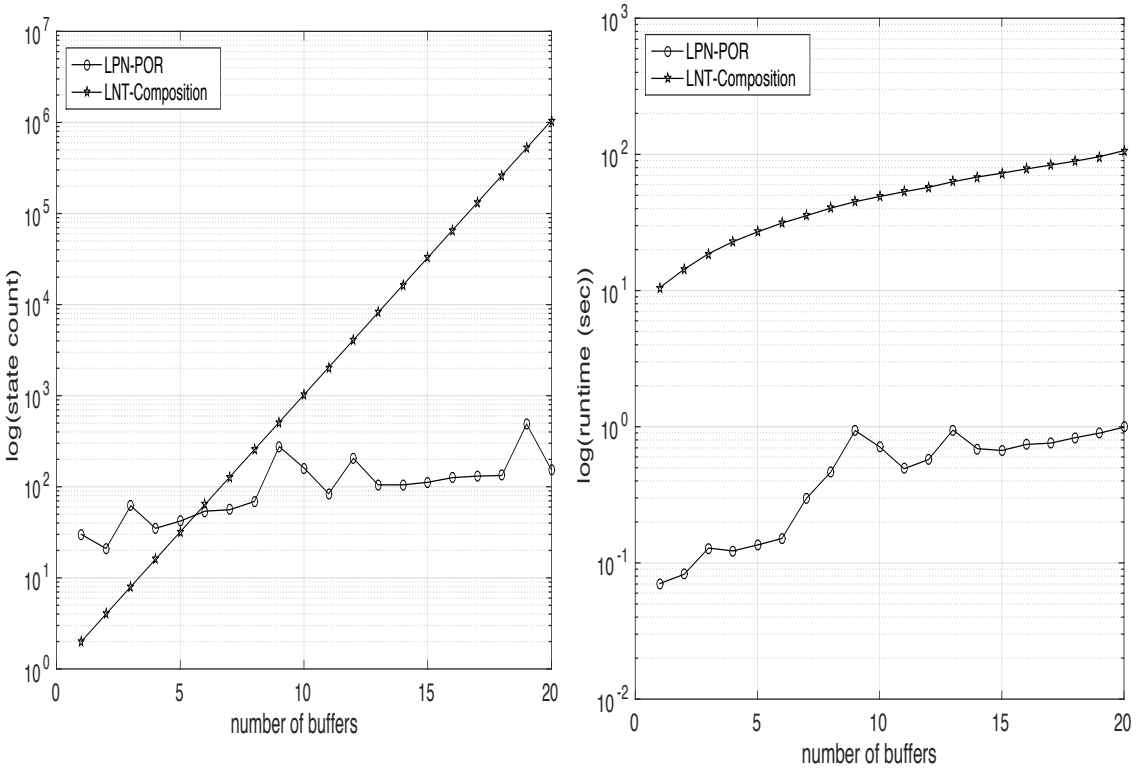


Figure 5.5: State count comparison between trace-back and behavioral analysis.

5.2.6 Comparisons Between POR-TB with Compositional Minimization

Comparisons are also made on state count, runtime and memory usage between the POR-TB approach in LEMA and the compositional minimization (CM) approach in CADP. The same set of buffer examples with asynchronous communication are used. Similar LNT and VHDL examples with two buffers in series have been presented in Figures 2.3 and 2.4 and Figures 2.13 to 2.16. The difference is that each buffer example used here does not nondeterministically assign different values in its producers, but rather uses an initial value of 0. All experiments have been performed on a Linux machine with a CPU of eight 3.60 GHz cores, and 16GB of available RAM is used to generate the results in this section. One core is used at any time for all experiments conducted with both tools. Note that no transition labels are hidden for any LNT models.

Figure 5.6 shows comparisons of state counts and medians of 50 runtimes on a logarithmic scale between the two approaches. The POR-TB approach produces a near-linear increase of state counts as more buffers are inserted, while the CM approach produces an exponential increase. The former's runtimes are two orders of magnitude shorter than the latter's.



(a) State count comparison. (b) Runtime comparison.

Figure 5.6: State counts and runtimes comparisons of the buffer examples with 1 to 20 buffers.

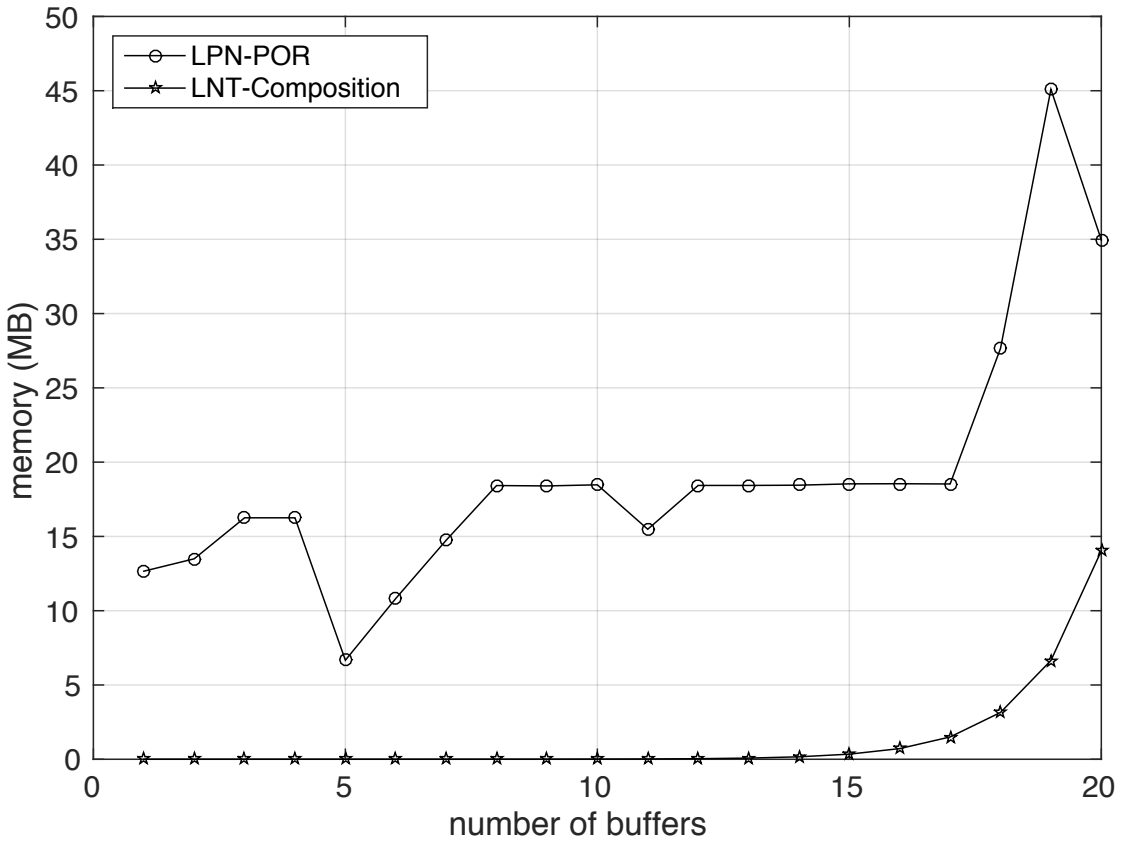


Figure 5.7: Memory comparison of the buffer examples with 1 to 20 buffers.

For the memory usage comparison shown in Figure 5.7, the CADP tool is more advantageous. Even though the state counts produced by the LNT models are significantly larger, their corresponding memory footprint is kept small. The difference in memory usage is largely due to different language implementations: LEMA uses Java which relies on its garbage collection for memory management, whereas CADP is implemented in C where it optimizes state space storage with full control of memory management.

It is worth noting that the series buffer examples are loosely coupled with little transition dependencies. Indeed, POR approaches have distinct advantages in handling systems with high concurrency but low dependencies. On the other hand, compositional minimization used to conduct these experiments minimizes LTSs with respect to divergence-sensitive branching bisimulation equivalence. The need for preserving branching structures may prevent significant state minimizations. Also, transition label hiding plays a key role for composing and minimizing LTSs. State reduction can improve significantly as more labels are hidden. Actually, with all labels hidden in each buffer example in the series, they all

reduce down to a final state space with a single state and a single transition. Runtimes, however, show only slight improvement over the case where all labels are visible, and are still two orders of magnitude longer than those produced by POR-TB. The runtime overhead is because the CM approach is not performed by a single tool, but that SVL generates a shell script, which generates quite a lot of tool invocations: for instance, each composition step requires at least a call to choose the set of processes to combine, a call to the composition tool, and a call to the minimization tool, not to mention the generation and clean up of auxiliary files.

5.3 VHDL Model for the Two-by-Two NoC

The livelock-free routing architecture for the two-by-two NoC in Figure 4.13 has been modeled in the channel-level VHDL. This section describes several representative components of the routers and arbiters with their VHDL descriptions.

5.3.1 Modeling Nondeterministic Choice in Arbiters

The behavior of nondeterministic choice is needed for all arbiters in the model. Figure 5.8 shows the entity for an arbiter with two inputs. This arbiter is instantiated by all arbiters except the PE arbiters and arbiters making an illegal turn. Port declarations are omitted from the example code. This arbiter uses two pairs of ports to handle communications with a PE router and a non-PE one: (in0_status, in0) and (in1_status, in1). Ports with the "_status" suffix are used to inform the connected routers of the arbiter's output status: a Boolean value 1 indicates a nonfaulty output link, and 0 indicates a faulty link. The output port arb_out is used by the arbiter to send data, i.e., one_flit, to its connected succeeding router. All ports are all of type *channel* , and are initialized by the function call *init_ channel*.

The behavior of this arbiter is that it blocks until either in0_status or in1_status channel requests to receive its output link status. If both requests are sampled by their respective probes simultaneously, variable z is nondeterministically assigned to a value of 1 or 2, based on which the corresponding channel is selected by the arbiter to receive data. This data is then sent out on the arb_out channel. On the other hand, if a request is detected only on one of the channels, the arbiter communicates on that channel with the connected router in a similar way. All PE arbiters have a slightly simpler behavior in that they receive the data directly without reporting the output link status first, as they do not have any outputs and only consume data packets.

For an arbiter that helps to make an illegal turn (e.g. arb_W_10, arb_S_11), it needs to be able to send negative acknowledgements to the router. The snippet of a two-input

```
architecture behavior of arb_2_in is
  signal one_flit : std_logic_vector(1 downto 0);
  signal sig_one : std_logic_vector(0 downto 0) := "1";
begin
  arb_2_in : process
    variable z : integer;
  begin
    await_any(in0_status, in1_status);
    vassign(sig_one,"1",1,1);
    if (probe(in0_status) and probe(in1_status)) then
      z := selection(2);
      if (z = 1) then
        send(in0_status, sig_one);
        receive(in0, one_flit);
        send(arb_out, one_flit);
      else
        send(in1_status, sig_one);
        receive(in1, one_flit);
        send(arb_out, one_flit);
      end if;
    elsif (probe(in0_status)) then
      send(in0_status, sig_one);
      receive(in0, one_flit);
      send(arb_out, one_flit);
    else
      send(in1_status, sig_one);
      receive(in1, one_flit);
      send(arb_out, one_flit);
    end if;
  end process arb_2_in;
end behavior;
```

Figure 5.8: VHDL entity for the arbiter with two inputs.

buffer's behavior, i.e., the behavior when both of its input channels request to communicate, is shown in Figure 5.9. Similar to the arbiter in Figure 5.8, it arbitrarily selects one router to communicate with. The difference is that it has to send a negative acknowledgement so that the router attempting an illegal turn can drop the packet to avoid deadlock. The behavior is modeled with a breakable *while-loop*. A standard logic variable free is used to model the arbiter's availability to receive a network flit. After receiving a packet from its connected PE router on in_pe_router (Line 6), the arbiter assigns free to 0 with a uniform delay of one time unit (Line 7), indicating the arbiter is occupied. The *vassign* statement is defined in the *handshake* package [66]. Each iteration of the *while-loop* L1 (Lines 8 to 24) first waits until either the router attempting an illegal turn requests the arbiter's status (in_illegal_status) or the router connected to the arbiter's output is ready to receive its data flit (arb_out) (Line 9). Nondeterministic choice is made when both requests are sampled simultaneously (Lines 10 to 17). After the arbiter successfully sends out its flit on arb_out

```
1      await_any(in_pe_router_status, in_illegal_status);
2      if (probe(in_pe_router_status) and probe(in_illegal_status)) then
3        z0 := selection(2);
4        if (z0 = 1) then
5          send(in_pe_router_status, sig_one);
6          receive(in_pe_router, one_flit);
7          vassign(free, '0', 1, 1);
8          L1: while (free = '0') loop
9            await_any(in_illegal_status, arb_out);
10           if (probe(in_illegal_status) and probe(arb_out)) then
11             z1 := selection(2);
12             if (z1 = 1) then
13               send(arb_out, one_flit);
14               vassign(free, '1', 1, 1);
15             else
16               send(in_illegal_status, sig_zero);
17             end if;
18           elsif (probe(in_illegal_status)) then
19             send(in_illegal_status, sig_zero);
20           else
21             send(arb_out, one_flit);
22             vassign(free, '1', 1, 1);
23           end if;
24         end loop L1;
25       else
26         send(in_illegal_status, sig_one);
27         receive(in_illegal, one_flit);
28         vassign(free, '0', 1, 1);
29         L2: while (free = '0') loop
30           await_any(in_illegal_status, arb_out);
31           if (probe(in_illegal_status) and probe(arb_out)) then
32             z1 := selection(2);
33             if (z1 = 1) then
34               send(arb_out, one_flit);
35               vassign(free, '1', 1, 1);
36             else
37               send(in_illegal_status, sig_zero);
38             end if;
39           elsif (probe(in_illegal_status)) then
40             send(in_illegal_status, sig_zero);
41           else
42             send(arb_out, one_flit);
43             vassign(free, '1', 1, 1);
44           end if;
45         end loop L2;
46       end if;
```

Figure 5.9: Partial VHDL entity for the two-input arbiter with negative acknowledgement.

(Line 13), it sets free to 1 (Line 14) so that it can exit loop L1, after which the arbiter is ready to accept the next flit. If the arbiter chooses to communicate with the router at its input (Line 15), it sends a negative acknowledgement represented by sig_zero (a Boolean signal initialized to 0) on channel in_illegal_status (Line 16). This models the behavior that this arbiter is not available to receive a flit and tells the router to drop its flit to avoid deadlock. It repeats sending a negative acknowledgement, indicated by the free variable remaining at 0, until it clears out its own flit. Note that both a busy and a faulty arbiter output link are modeled with the same signal value 0. Differentiating them is unnecessary because in either case, the router receiving this status has to drop its packet. The rest of this arbiter model uses the same control structures.

5.3.2 Router Models

Each of the PE routers has similar behaviors: it first nondeterministically assigns a two-bit destination coordinate to a flit, and then checks each bit against its own location coordinates, the result of which determines the next forwarding location for the flit. Figure 5.10 describes the PE router of node 10. It has two pairs of output ports: (out_w_status, out_w) and (out_n_status, out_n). They are all typed *channel*, and are initialized by the function called *init_channel*.

The random choice in a nondeterministic data assignment is modeled with a variable z that can take a random value, and different destinations are assigned in each choice branch (Lines 4 to 11). The expression one_flit(0 **downto** 0) = "0" (Line 12) compares the horizontal coordinate of one_flit to the destination coordinate 0. For a packet destined for either node

```
1   r_PE_10 : process
2     variable z : integer;
3   begin
4     z := selection(3);
5     if (z = 1) then
6       vassign(one_flit,"00",1,1);
7     elsif (z = 2) then
8       vassign(one_flit,"01",1,1);
9     else
10      vassign(one_flit,"11",1,1);
11    end if;
12    if (one_flit(1 downto 1) = "0") then
13      receive(out_w_status, arb_status);
14      if (arb_status = "0") then
15        receive(out_n_status, arb_status);
16        if (arb_status = "0") then
17          vassign(fail_to_route, '1', 0, 0);
18          guard(fail_to_route, '0');
19          vassign(fail_to_route, '0', 0, 0);
20        else
21          send(out_n, one_flit);
22        end if;
23      else
24        send(out_w, one_flit);
25      end if;
26    elsif (one_flit(0 downto 0) = "1" and one_flit(1 downto 1) = "1") then
27      receive(out_n_status, arb_status);
28      if (arb_status = "0") then
29        receive(out_w_status, arb_status);
30        if (arb_status = "0") then
31          vassign(fail_to_route, '1', 0, 0);
32          guard(fail_to_route, '0');
33          vassign(fail_to_route, '0', 0, 0);
34        else
35          send(out_w, one_flit);
36        end if;
37      else
38        send(out_n, one_flit);
39      end if;
40    end if;
41  end process r_PE_10;
```

Figure 5.10: VHDL entity for the NoC PE router of node 10.

00 or 01, the router first tries to send the packet west (Line 13). If this direction is not possible due to a faulty link (Line 14), it diverts the packet to its north output (Line 15). Otherwise it forwards the packet to its west output (Line 25). If the north direction is not viable after the packet is diverted, then a routing failure occurs, which is modeled by assigning the standard logic signal fail_to_route to 1 (Line 17). The following guard statement (Line 18) only executes if fail_to_route evaluates to 0. Since this signal is just assigned to 1, it will not finish executing this line, which creates a deadlock. This is useful for the verification tool to detect routing failure. The next line (Line 19) is vacuous as it is never reachable. Its existence is to pair with the previous assignment to the same signal (Line 17) to facilitate correct compilations of this VHDL model. On the other hand, if the north direction is not faulty, the packet is sent to north (Line 21).

Unlike the PE router, a non-PE router receives data flits instead of generating them. Some non-PE routers are also responsible for dropping flits. In Figure 5.11, router r_W_10 drops its flit after a failure attempt to send it back west (Line 11). It is an illegal turn because this router only receives east-going packets, and hence making a west turn is illegal. Also, note that if a packet's destination is node 01, the router does not try an alternative path after it fails to send it north (Lines 20 to 29). This is due to the removal of multiple diversions to avoid livelock.

5.4 Verification Observations

Auxiliary VHDL entities are provided to model the environment that produces and/or consumes network packets for each router and arbiter. A LPN is automatically compiled from the VHDL specifications for each router or arbiter and their respective environment. Full state exploration on these individual models produces large state spaces due to the existence of high concurrency and nondeterminism. The full state search for the LPN model of the arbiter with two inputs, whose VHDL entity is shown in Figure 5.8, generated 713,122 states. POR-TB has demonstrated effective state reduction on the two-input arbiter with negative acknowledgement, whose representative VHDL entity is shown in Figure 5.9, and the PE arbiter: it reduces the state count for the former from 5,778 to 81, and the latter from 20,192 to 1795, achieving over 90% state reduction. Unfortunately, due to a glitch in the POR-TB Java implementation, state reduction on other routers and arbiters ran into erroneous deadlock states.

Compared to the state count generated from the corresponding LNT processes, the LPN exhibits significant overhead, even after POR-TB is applied. For example, state count for the

```
1   r_W_10: process
2   begin
3       receive(in_w, one_flit);
4       if (one_flit(0 downto 0) = "0" and one_flit(1 downto 1) = "1") then
5         send(out_ip, one_flit);
6       elsif (one_flit(0 downto 0) = "1" and one_flit(1 downto 1) = "1") then
7         receive(out_n_status, arb_status);
8         if (arb_status = "0") then
9           receive(out_w_status, arb_status);
10          if (arb_status = "0") then
11            vassign(drop, '1', 1, 1);
12            guard(drop, '0');
13            vassign(drop, '0', 0, 0);
14          else
15            send(out_w, one_flit);
16          end if;
17        else
18          send(out_n, one_flit);
19        end if;
20      elsif (one_flit(0 downto 0) = "1" and one_flit(1 downto 1) = "0") then
21        receive(out_n_status, arb_status);
22        if (arb_status = "0") then
23          vassign(rm_livelock, '1', 0, 0);
24          guard(rm_livelock, '0');
25          vassign(rm_livelock, '0', 0, 0);
26        else
27          send(out_n, one_flit);
28        end if;
29      end if;
30  end process r_W_10;
```

Figure 5.11: VHDL entity for the west router of node 10.

two-input arbiter with negative acknowledgement and the PE arbiter is 7 with 18 transitions and 1 with 8 transitions. This discrepancy is largely due to different representations of synchronization operations in LNT and LPN. LNT encodes a pair of "send" and "receive" operations on the same channel with a pair of synchronization gates that exchange offers. CADP generates a single-labeled transition and one state on the LTS when the rendezvous of this pair of gates happens. LPN, however, uses six transitions to represent the handshake between these operations without an explicit notion of synchronization, as shown in Figure 5.2. As can be seen from Figure 5.3, full state exploration in LEMA interleaves all possible transition-firing sequences, producing 16 states. Even after the removal of unnecessary interleavings by POR-TB, there are still 10 states. Other VHDL compilation overhead, such as a dummy LPN transition for each branch of a nondeterministic assignment, and complex LPN enabling condition expressions for standard logic vector slicing, has also contributed to the large state space and slow runtime.

5.5 Conclusion and Discussion

This chapter first presents the evaluation of the costs and benefits of using trace-back to improve the ample set computation for partial order reduction on several asynchronous designs. For examples with significant concurrency, such as the series buffers examples and large FIFO circuit models, trace-back can produce better ample sets, leading to a significant reduction in the number of states that must be explored during verification. This result can provide both memory and runtime benefits. The asynchronous circuit models are particularly challenging for this approach due to the fact that every transition in these models is a disabling failure transition. However, with the proposed refinements in the preparations step (Section 5.2.1), our method achieves considerable state reduction. This is largely due to the fact that the proposed refinements statically eliminate a large amount of seemingly dependent transitions, avoiding unnecessary dependency analysis for the state exploration. It should also be noted that, in general, one cannot expect a partial order reduction method to be effective for *every* model. It tends to work well on models that have components are loosely coupled, giving more concurrency than dependency. This can be seen from the state count comparisons between POR-TB and the compositional minimization on the series of buffers. The buffer examples are ideal cases for POR-TB, as transitions have little dependency but substantial concurrency. Another key observation is that gate hiding in LNT plays a very important role on compositional minimization. It is somewhat related to partial order reductions, in the sense that irrelevant interleavings of internal transitions are abstracted away. Representative VHDL models are described for the routers and arbiters of the livelock-free two-by-two NOC. They are compiled to LPN models, which are used by LEMA to do state exploration. POR-TB enables over 90 percent state reduction over the full state space on two arbiter models. LNT, which uses explicit synchronizations, still outperforms POR-TB. Observations show that the main reasons for this are the lack of LPN-explicit synchronization and inefficient compilation of some VHDL constructs and expressions. One possible solution to the implicit LPN synchronization is to apply state minimizations on immediate states, which are generated by immediate transitions during state exploration, to only keep a single, nonimmediate state in a synchronization operation. VHDL compilation also needs to be improved to achieve more direct and efficient conversion to LPN.

CHAPTER 6

CONCLUSION

A NoC approach to flexible mapping of Ecus to sensors and actuators on an automotive system has the following distinctive advantages: it is possible for Ecus to balance their computation load by sharing processing power through network communications; it provides spare control units that can replace the faulty ones without losing control of their designated sensors and/or actuators. It is desirable to implement a fault-tolerant NoC routing algorithm to provide additional resilience to network faults. Fault-tolerance, however, increases the design complexity, making the NoC routing prone to deadlock and other problems. This dissertation proposes a link-fault-tolerant routing algorithm and its formal models that are verified to prove its functional correctness. This chapter concludes the dissertation by providing a summary and presenting possible future work.

6.1 Summary

This dissertation describes the development of a link-fault-tolerant routing algorithm on an asynchronous, multiflit routing architecture. It is formally modeled in a process-algebra action-based language Lnt, and a channel-level Vhdl which can be automatically compiled to a transition-based Lpn formalism. Compositional verification has been applied to the Lnt model to successfully prove several desirable functional properties. An optimal Por approach is proposed to ease the combinatorial state explosion for state reachability of Lpns. With the help of this approach, significant state reduction is achieved on two arbiters of the two-by-two NoC model. Key observations are made between the two competing verification methodologies.

Improving upon the node-fault-tolerant Glass/Ni routing algorithm, the proposed routing algorithm presented in Chapter 3 loosens the stringent and unrealistic node-fault assumption, and is able to achieve link-fault tolerance on a 2-D mesh network. To guarantee deadlock freedom, this algorithm drops network packets to avoid formations of dependency cycles. The cost of this performance tradeoff is shown to be minimal from the simulation results of a Vhdl implementation of the routing algorithm. A routing architecture is

proposed to improve efficiency by allowing simultaneous routing of multiple packets on a single node.

Evolution of the LNT model for this NoC architecture is described in Chapter 4. Key lessons learned from a leakage path and a deadlock scenario after removing all arbiters' buffering capacity have contributed to the construction of a correct behavioral model for the arbiters. To address a major source of state explosion, a data abstraction scheme is proposed that maps a flit's concrete destination coordinates to its diversion status. The discovery of routing failure hidden by the abstract model leads to the discovery of excessive fault-tolerance in the routing algorithm that can cause livelock issues. Removal of livelock loops yields a simplified architecture that enables full state space generation of the concrete model. With the help of the CADP verification toolbox, functional properties are verified for deadlock and livelock freedom and packet delivery.

The obtained livelock-free two-by-two NoC architecture is then modeled in channel-level VHDL, from which the LPN model can be generated and used for verification. Representative VHDL entities of this NoC archietecture are described in Chapter 5. Given the highly concurrent nature of this NoC model, a different avenue of state reduction technique, namely partial order reduction, is studied in detail. Algorithms are proposed for a minimal ample set construction method with tracing back on processes of the LPN model to obtain transitions necessary to interleave, while pruning certain seemingly dependent transitions. Evaluation of the costs and benefits of using partial order reduction with trace-back on several asynchronous designs shows that this mechanism works well on models that have loosely coupled components that exhibit more concurrency than dependency. It manages to substantially reduce the state space of two arbiter models of the two-by-two NoC. However, the state spaces are still larger than the corresponding LNT models that use explicit synchronization. Analysis of this difference suggests the need for minimizing states created by LPN immediate transitions and improving the VHDL compiler.

6.2 Future Work

Experiences and insights gained in using formal analysis techniques to debug and verify properties of the complex NoC routing architecture have inspired us to pursue several future research directions. This section describes some potential approaches to undertaking these research investigations.

6.2.1 Large-scale NoC Verification Using CADP

Although it is possible to obtain successful verification results for the LNT version of the two-by-two NoC, as described in Section 4.4, the two-by-two NoC only includes four corner nodes that do not exhibit all possible behaviors specified by the link-fault routing algorithm. On the other hand, the three-by-three NoC, as shown in Figure 3.11, contains all the nine different types of nodes that can exercise all possible behaviors of the routing algorithm specified in Figure 4.14. Unfortunately, deadlock freedom verification on a purely abstract version of this NoC fails due to state explosion. Other general state reduction and optimization techniques are considered, but they do not seem to be promising in dealing with the large NoC example. Detailed remarks are given below.

- Symmetry reduction. The considered three-by-three NoC is not quite symmetrical, as can be seen from its architecture in Figure 4.13. Added upon it is the nonsymmetrical negative-first-based routing algorithm that gives bias to negative routing directions and sets special routing rules on the two negative edges. Therefore, applying symmetry reduction may not be effective.

- Assign priorities to transitions. This technique could be potentially useful, but priorities are incompatible with the compositional approach, because the divergence-sensitive branching bisimulation is not a congruence for priority. Though priorities would work for strong bisimulation, which would enable far fewer reductions.

- Symbolic techniques. CADP does not yet support them. But but for such an asymmetric system, the benefit of symbolic techniques is not completely obvious.

- Iterative techniques (e.g., Counter-Example Guided Abstraction Refinement (CEGAR), or BMC techniques) are also not guaranteed to succeed. To the best knowledge of the author, data abstraction adopted in this work is the coarsest, i.e., only a one-bit Boolean variable remains to indicate the diversion status. Even with the coarsest abstraction, state explosion still occurs. It is not clear if the CEGAR approach can result in more significant abstraction, and any refinement on the abstraction is expected to worsen the already unmanageable state space. As for BMC techniques, it may be difficult, if possible, to check whether a deadlock state exists without a full state space. Also, it has been applied by Palaniveloo [112] without success.

The prevailing asymmetry in the three-by-three NoC poses significantly challenging tasks, especially for deadlock freedom verification where all routing nodes continuously

generate network packets. One idea to tackle this problem is to avoid generating the global state space (or more precisely, LTS), but to reason about deadlock from locally generated LTSs. Partial model checking [149] is potentially a promising option. Rooted in the compositional verification, this technique incrementally incorporates the extracted behavior from a process into a temporal logic formula to obtain a new formula, and this formula can be verified on a composition of the rest of the processes, i.e., all processes excluding the one already incorporated in the formula. Simplifications of the formula must be applied to keep its size tractable. Recently, Lang and Mateescu [150] generalized and optimized this technique to the network of LTSs and implemented as a companion software for the CADP toolbox. Another idea worth exploring is to experiment with techniques using state space caching, i.e., not all states are stored, but only some.

6.2.2 Combining Static Analysis with Dynamic Analysis

While model checking is needed to check implementation details, such as connectivity of routers and arbiters; and dynamic properties, such as deadlock avoidance; static analysis methods such as those used by DCI2 and GeNoC can be much more efficient.

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

Therefore, an interesting area of future research is to develop methodologies that leverage both techniques. For example, DCI2 can be used to refine a routing protocol to eliminate

static deadlocks, livelocks, and disconnected route conditions. Once the routing protocol is developed, model checking can be employed to check the implementation architecture and verify dynamic properties.

6.2.3 Improving Partial Order Reduction on LPNs

The cost and benefit evaluation of using trace-back of POR on LPNs discussed in Section 5.2.5 has shown a performance penalty in computing an optimal ample set at each state. There are several approaches to improve the results for POR with trace-back. First, limiting the depth of the necessary set calculation can reduce the run time at perhaps some loss of optimality. Second, the calculation of our `DisabledBy` and `Disable` sets is done in a fairly conservative fashion. Namely, if a transition assigns to any variable that can potentially disable an enabling condition, this transition is assumed to disable the transition. A better analysis of the enabling condition should be able to eliminate false disablings. Since this is statically computed at the beginning of analysis, it is a one-time-only cost, and its complexity is on the size of the model and not the size of the state space.

Another future research direction to be investigated is to leverage advantages of different state reductions. Experiences gained from this work have demonstrated that a particular state reduction technique is good at handling one type of features. For example, data abstraction is effective for models with complex data structures, whereas POR is mostly efficient for highly concurrent models. However, for many real-world systems, it is often the case that they possess several features, which degrades the effectiveness of applying a single state reduction method. It is therefore necessary to explore ideas that intelligently combine several state reduction methods. Not only can a combined approach improve the state reduction efficiency, but also each individual technique can potentially benefit from others. One possible approach is to use local state graphs generated by compositional verification approach to infer more accurate trace-back for POR. The basic idea is to generate local state using the expansion method presented by Zheng [151], then use local state information to remove unnecessary concurrency. Depending on the quality of the generated local state graphs, removing concurrency based solely on the local state information may not always be efficient [152], especially when the dependency information is not obvious, i.e., for each enabled transition, it is dependent on at least one other transition that is not enabled. A naive approach is to take the enabled set as ample set in this case. However, it is believed that if trace-back is applied in this case, more refined dependency relation for each enabled transition can be obtained, and the resultant ample set can potentially be smaller. Also,

due to the limited usage of trace-back in this combined approach, the performance cost can decrease for the combined approach.

6.2.4 Stochastic Analysis

Packet drop can limit the effectiveness of the routing algorithm, especially when a particular link is faulty, as can be seen from Table 4.2. Packet drop, however, is necessary for deadlock avoidance in our link-fault routing algorithm. It is a challenging task to justify the performance of the routing algorithm in terms of packet drop due to deadlock avoidance in a pure functional verification setting. To obtain a more accurate justification, it is promising to annotate LTS transitions obtained for the functional analysis with link failure probabilities, and then apply numerical quantitative methods such as the *Continuous-Time Markov Chain* (CTMC) analysis to evaluate its performance.

One widely known difficulty for the CTMC analysis is also state explosion. This is because, in general, Markovian analysis requires a full state space of a system to perform calculations. A possible research direction is to apply POR techniques on stochastic models for the CTMC analysis. There are two foreseeable challenges here. Firstly, for a nonprobabilistic model, the ample set is always chosen to be the set with the smallest number of necessary enabled transitions. This may not be the case for CTMCs. When there is a choice for determining an ample set, transition rates will have to be considered. Improperly selected state-transition sequence by a POR technique may have the least significant accumulated probability, leading to the enabling of a failure transition, which should not be missed. The second challenge is the transition rates approximation. Pruning transition firings does not change the rates on the remaining ones. However, when performing transient analysis on CTMCs, one standard technique is to use uniformisation, which involves converting CTMCs to embedded *Discrete-Time Markov Chains* (DTMCs). One step in this conversion requires rates-to-probabilities transformation, and the pruned rates are converted to probabilities of staying at a state. Therefore, POR on CTMCs can potentially decrease the probability of leaving a state for any remaining transition. The rates on the remaining transitions have to be approximated. It is possible to produce a uniformly distributed range of probabilities where lower and upper bounds are derived after POR prunes transitions. However, this idea needs to be further investigated to examine what types of probabilistic properties can be checked with reasonable accuracy and whether efficient algorithms can be derived to perform CTMC analysis on a range of probabilities.

REFERENCES

[1] E. A. Lee, "Cyber Physical Systems: Design Challenges," tech. rep., Center for Hybrid and Embedded Software Systems, EECS University of California, Berkeley Berkeley, CA 94720, USA, 2008.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking.* MIT Press, 1999.

[3] A. Valmari, *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, ch. The state explosion problem, pp. 429–528. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10ˆ20 states and beyond," *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.

[5] K. L. McMillan, *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[6] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, pp. 677–691, Aug. 1986.

[7] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, 1994.

[8] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings*, ch. Symbolic Model Checking without BDDs, pp. 193–207. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.

[9] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.

[10] A. Pnueli, *Logics and Models of Concurrent Systems*, ch. In Transition From Global to Modular Temporal Reasoning about Programs, pp. 123–144. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985.

[11] E. Clarke, D. Long, and K. McMillan, "Compositional model checking," in *Proceedings of the Fourth Annual Symposium on Logic in computer science*, (Piscataway, NJ, USA), pp. 353–362, IEEE Press, 1989.

[12] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28 – July 2, 1998 Proceedings*, ch. You assume, we guarantee: Methodology and case studies, pp. 440–451. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[13] S. Graf, B. Steffen, and G. Lüttgen, "Compositional Minimisation of Finite State Systems Using Interface Specifications," *Formal Asp. Comput.*, vol. 8, no. 5, pp. 607–616, 1996.

[14] D. Bustan, "Modular minimization of deterministic finite-state machines," in *In 6th International Workshop on Formal Methods for Industrial Critical Systems*, pp. 163–178, 2001.

[15] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 1512–1542, Sept. 1994.

[16] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM Trans. Program. Lang. Syst.*, vol. 19, pp. 253–291, Mar. 1997.

[17] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting symmetry in temporal logic model checking," *Form. Methods Syst. Des.*, vol. 9, pp. 77–104, Aug. 1996.

[18] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," *Form. Methods Syst. Des.*, vol. 9, pp. 105–131, Aug. 1996.

[19] C. N. Ip and D. L. Dill, "Better verification through symmetry," *Form. Methods Syst. Des.*, vol. 9, pp. 41–75, Aug. 1996.

[20] D. Peled, *Computer Aided Verification: 5th International Conference, CAV '93 Elounda, Greece, June 28–July 1, 1993 Proceedings*, ch. All from one, one for all: on model checking using representatives, pp. 409–423. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993.

[21] G. J. Holzmann and D. Peled, "An improvement in formal verification," in *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, (London, UK), pp. 197–211, Chapman & Hall, Ltd., 1995.

[22] A. Valmari, "A stubborn attack on state explosion," *Formal Methods in System Design*, vol. 1, no. 4, pp. 297–322.

[23] A. Valmari, *Computer Aided Verification: 5th International Conference, CAV '93 Elounda, Greece, June 28–July 1, 1993 Proceedings*, ch. On-the-fly verification with stubborn sets, pp. 397–408. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993.

[24] P. Godefroid, *Computer-Aided Verification: 2nd International Conference, CAV '90 New Brunswick, NJ, USA, June 18–21, 1990 Proceedings*, ch. Using partial orders to improve automatic verification methods, pp. 176–185. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991.

[25] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer Berlin Heidelberg, 1996.

[26] E. A. Emerson, S. Jha, and D. Peled, *Tools and Algorithms for the Construction and Analysis of Systems: Third International Workshop, TACAS'97 Enschede, The Netherlands, April 2–4, 1997 Proceedings*, ch. Combining partial order and symmetry reductions, pp. 19–34. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997.

[27] A. Valmari, "Stubborn Sets of Coloured Petri Nets.," in *Proceedings of the 12th International Conference on Application and Theory of Petri Nets, 1991, Gjern, Denmark*, pp. 102–121, June 1991.

[28] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-order reduction in symbolic state-space exploration," *Form. Methods Syst. Des.*, vol. 18, pp. 97–116, Mar. 2001.

[29] H. Hansen and X. Wang, "Compositional Analysis for Weak Stubborn Sets," in *Proceedings of the 2011 Eleventh International Conference on Application of Concurrency to System Design*, ACSD '11, (Washington, DC, USA), pp. 36–43, IEEE Computer Society, 2011.

[30] I. Konnov, H. Veith, and J. Widder, "On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability," in *CONCUR 2014 — Concurrency Theory* (P. Baldan and D. Gorla, eds.), vol. 8704 of *Lecture Notes in Computer Science*, pp. 125–140, 2014.

[31] I. Konnov, H. Veith, and J. Widder, "SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms," in *CAV (Part I)*, vol. 9206 of *LNCS*, pp. 85–102, 2015.

[32] C. J. Glass and L. M. Ni, "Fault-Tolerant Wormhole Routing in Meshes," in *Twenty-third annual international symposium on falut=tolerant computing*, pp. 240–249, 1993.

[33] J. Wu, Z. Zhang, and C. Myers, "A Fault-Tolerant Routing Algorithm for a Network-on-Chip Using a Link Fault Model," in *Virtual Worldwide Forum for PhD Researchers in Electronic Design Automation*, 2011.

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

[35] Z. Zhang, W. Serwe, J. Wu, T. Yoneda, H. Zheng, and C. Myers, *Formal Methods for Industrial Critical Systems: 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, ch. Formal Analysis of a Fault-Tolerant Routing Algorithm for a Network-on-Chip, pp. 48–62. Cham: Springer International Publishing, 2014.

[36] Z. Zhang, W. Serwe, J. Wu, T. Yoneda, H. Zheng, and C. Myers, "An improved fault-tolerant routing algorithm for a network-on-chip derived with formal analysis," *Science of Computer Programming*, vol. 118, pp. 24 – 39, 2016. Formal Methods for Industrial Critical Systems (FMICS'2014).

[37] R. A. Thacker, K. R. Jones, C. J. Myers, and H. Zheng, "Automatic abstraction for verification of cyber-physical systems," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, (New York, NY, USA), pp. 12–21, ACM, 2010.

[38] R. A. Thacker, *A New Verification Method for Embedded Systems*. Ph.D. dissertation, Sch. Comp., Univ. of Utah, Salt Lake City, UT, Jan. 2010.

[39] G. D. Plotkin, "The origins of structural operational semantics," *The Journal of Logic and Algebraic Programming*, vol. 60–61, no. 0, pp. 3 – 15, 2004. Structural Operational Semantics.

[40] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 1980.

[41] R. Milner, *Communication and Concurrency.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[42] R. Milner, *Communicating and Mobile Systems: The π-calculus.* New York, NY, USA: Cambridge University Press, 1999.

[43] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *J. ACM*, vol. 31, no. 3, pp. 560–599, 1984.

[44] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[45] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[46] A. J. Martin, "Developments in concurrency and communication," ch. Programming in VLSI: From Communicating Processes to Delay-insensitive Circuits, pp. 1–64, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[47] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proceedings of the Conference on European Design Automation*, EURO-DAC '91, (Los Alamitos, CA, USA), pp. 384–389, IEEE Computer Society Press, 1991.

[48] J. Bergstra and J. Klop, "Process algebra for synchronous communication," *Information and Control*, vol. 60, no. 1–3, pp. 109 – 137, 1984.

[49] ISO/IEC, "Lotos — a formal description technique based on the temporal ordering of observational behaviour." International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[50] K. J. Turner, *The Formal Specification Language* LOTOS*: A Course For Users*, Aug. 1989.

[51] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification I.* 6, Springer-Verlag Berlin Heidelberg, 1 ed., 1985.

[52] R. Mateescu and W. Serwe, *Formal Methods for Industrial Critical Systems: 15th International Workshop, FMICS 2010, Antwerp, Belgium, September 20-21, 2010. Proceedings*, ch. A Study of Shared-Memory Mutual Exclusion Protocols Using CADP, pp. 180–197. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[53] F. Ghassemi, W. Fokkink, and A. Movaghar, "Verification of mobile ad hoc networks: An algebraic approach," *Theoretical Computer Science*, vol. 412, no. 28, pp. 3262 – 3282, 2011. Festschrift in Honour of Jan Bergstra.

[54] R. Abid, G. Salaün, F. Bongiovanni, and N. Palma, *Automated Technology for Verification and Analysis: 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, ch. Verification of a Dynamic Management Protocol for Cloud Applications, pp. 178–192. Cham: Springer International Publishing, 2013.

[55] N. D. Mendes, F. Lang, Y. L. Cornec, R. Mateescu, G. Batt, and C. Chaouiya, "Composition and abstraction of logical regulatory modules: application to multicellular systems," *Bioinformatics*, vol. 29, no. 6, pp. 749–757, 2013.

[56] H. Garavel and F. Lang, "SVL: a Scripting Language for Compositional Verification," in *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems {FORTE}'2001*, pp. 377–392, Kluwer Academic Publishers, Aug. 2001.

[57] H. Zheng, Z. Zhang, C. Myers, E. Rodriguez, and Y. Zhang, "Compositional model checking of concurrent systems," *Computers, IEEE Transactions on*, vol. 64, pp. 1607–1621, June 2015.

[58] H. Zheng, E. Mercer, and C. J. Myers, "Modular verification of timed circuits using automatic abstraction," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 22, no. 9, pp. 1138–1153, 2003.

[59] H. Zheng, C. J. Myers, D. Walter, S. Little, and T. Yoneda, "Verification of timed circuits with failure-directed abstractions," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 403–412, 2006.

[60] T. Yoneda and C. J. Myers, "Synthesis of timed circuits based on decomposition," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 7, pp. 1177–1195, 2007.

[61] D. Walter, S. Little, C. J. Myers, N. Seegmiller, and T. Yoneda, "Verification of analog/mixed-signal circuits using symbolic methods," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 12, pp. 2223–2235, 2008.

[62] S. Little, D. Walter, C. J. Myers, R. A. Thacker, S. Batchu, and T. Yoneda, "Verification of analog/mixed-signal circuits using labeled hybrid petri nets," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 617–630, 2011.

[63] R. Thacker, C. Myers, K. Jones, and S. Little, "A new verification method for embedded systems," in *Proc. International Conference on Computer Design (ICCD).*, IEEE Computer Society Press, 2009.

[64] H. Kuwahara, C. J. Myers, M. S. Samoilov, N. A. Barker, and A. P. Arkin, "Automated abstraction methodology for genetic regulatory networks," vol. 4220, pp. 150–175, 2006.

[65] C. Madsen, Z. Zhang, N. Roehner, C. Winstead, and C. J. Myers, "Stochastic model checking of genetic circuits," *JETC*, vol. 11, no. 3, pp. 23:1–23:21, 2014.

[66] C. Myers, *Asynchronous Circuit Design*. John Wiley & Sons, 2001.

[67] A. J. Martin, "The probe: An addition to communication primitives," *Information Processing Letters*, vol. 20, no. 3, pp. 125 – 130, 1985.

[68] H. Zheng, "Specification and compilation of timed systems," Ph.D. dissertation, Dept. Elect. & Comp. Eng., Univ. of Utah, Salt Lake City, UT, June 1998.

[69] E. R. Peskin, *Protocol Selection, Implementation, and Analysis for Asynchronous Circuits*. Ph.D. dissertation, Dept. Elect. & Comp., Univ. of Utah, Salt Lake City, UT, Aug. 2002.

[70] H. Garavel, G. Salaün, and W. Serwe, "On the semantics of communicating hardware processes and their translation into LOTOS for the verification of asynchronous circuits with CADP," *Sci. Comput. Program.*, vol. 74, no. 3, pp. 100–127, 2009.

[71] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '00, (New York, NY, USA), pp. 250–256, ACM, 2000.

[72] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684–689, 2001.

[73] B. Coates, A. Davis, and K. Stevens, "The post office experience: designing a large asynchronous chip.," *Integration*, vol. 15, no. 3, pp. 341–366, 1993.

[74] E. Beigné, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An asynchronous NOC architecture providing low latency service and its multi-level design framework," in *11th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2005), 14-16 March 2005, New York, NY, USA*, pp. 54–63, IEEE Computer Society, 2005.

[75] P. Vivet, D. Lattard, F. Clermidy, E. Beigne, C. Bernard, Y. Durand, J. Durupt, and D. Varreau, "FAUST, an Asynchronous Network-on-Chip based Architecture for Telecom Applications," *Proc. 2007 Design, Automation and Test in Europe (DATE07)*, 2007.

[76] J. You, Y. Xu, H. Han, and K. S. Stevens, "Performance evaluation of elastic GALS interfaces and network fabric," *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 1, pp. 17 – 32, 2008. Proceedings of the Third International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS 2007).

[77] R. R. Dobkin, R. Ginosar, and A. Kolodny, "QNoC asynchronous router," *Integration, the VLSI Journal*, vol. 42, no. 2, pp. 103 – 115, 2009.

[78] Y. Thonnart, P. Vivet, and F. Clermidy, "A fully-asynchronous low-power framework for gals noc integration," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 33–38, March 2010.

[79] M. Horak, S. Nowick, M. Carlberg, and U. Vishkin, "A low-overhead asynchronous interconnection network for gals chip multiprocessors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, pp. 494–507, April 2011.

[80] M. Imai and T. Yoneda, "Improving Dependability and Performance of Fully Asynchronous On-chip Networks," in *Proceedings of the 2011 17th IEEE International Symposium on Asynchronous Circuits and Systems*, ASYNC '11, (Washington, DC, USA), pp. 65–76, IEEE Computer Society, 2011.

[81] D. Gebhardt, J. You, and K. S. Stevens, "Design of an energy-efficient asynchronous noc and its optimization tools for heterogeneous socs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 30, no. 9, pp. 1387–1399, 2011.

[82] D. Gebhardt, J. You, and K. S. Stevens, "Link pipelining strategies for an application-specific asynchronous noc," in *NOCS 2011, Fifth ACM/IEEE International Symposium on Networks-on-Chip, Pittsburgh, Pennsylvania, USA, May 1-4, 2011* [157], pp. 185–192.

[83] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, (New York, NY, USA), pp. 278–287, ACM, 1992.

[84] Y. M. Boura and C. R. Das, "Efficient fully adaptive wormhole routing in n-dimensional meshes," in *Proceedings of the 14th International Conference on Distributed Computing Systems, Poznan, Poland, June 21-24, 1994*, pp. 589–596, IEEE Computer Society, 1994.

[85] A. A. Chien and J. H. Kim, "Planar-adaptive routing: Low-cost adaptive networks for multiprocessors," *J. ACM*, vol. 42, pp. 91–123, Jan. 1995.

[86] R. Casado, A. Bermúdez, F. J. Quiles, J. L. Sánchez, and J. Duato, "A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 115–132, Feb. 2001.

[87] D. Fick, A. DeOrio, G. Chen, V. Bertacco, D. Sylvester, and D. Blaauw, "A highly resilient routing algorithm for fault-tolerant NoCs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, (3001 Leuven, Belgium), pp. 21–26, European Design and Automation Association, 2009.

[88] R. Boppana and S. Chalasani, "Fault-tolerant wormhole routing algorithms for mesh networks," *Computers, IEEE Transactions on*, vol. 44, pp. 848–864, Jul 1995.

[89] C. Su and K. G. Shin, "Adaptive fault-tolerant deadlock-free routing in meshes and hypercubes," *IEEE Trans. Computers*, vol. 45, no. 6, pp. 666–683, 1996.

[90] J. Zhou and F. C. M. Lau, "Adaptive fault-tolerant wormhole routing in 2d meshes," in *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, April 23-27, 2001*, p. 56, IEEE Computer Society, 2001.

[91] J. Duato, "A theory of fault-tolerant routing in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, pp. 790–802, Aug. 1997.

[92] N. A. Nordbotten, M. E. Gómez, J. Flich, P. López, A. Robles, T. Skeie, O. Lysne, and J. Duato, *Network and Parallel Computing: IFIP International Conference, NPC 2004, Wuhan, China, October 18-20, 2004. Proceedings*, ch. A Fully Adaptive Fault-Tolerant Routing Methodology Based on Intermediate Nodes, pp. 341–356. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[93] K. Chen and G. Chiu, "Fault-tolerant routing algorithm for meshes without using virtual channels," *J. Inf. Sci. Eng.*, vol. 14, no. 4, pp. 765–783, 1998.

[94] T. T. Ye, L. Benini, and G. De Micheli, "Packetization and routing analysis of on-chip multiprocessor networks," *J. Syst. Archit.*, vol. 50, pp. 81–104, Feb. 2004.

[95] M. Li, Q.-A. Zeng, and W.-B. Jone, "Dyxy: A proximity congestion-aware deadlock-free dynamic routing method for network on chip," in *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, (New York, NY, USA), pp. 849–852, ACM, 2006.

[96] I.-G. Lee, J. Lee, and S.-C. Park, "Adaptive routing scheme for noc communication architecture," in *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on*, vol. 2, pp. 1180–1184, 2005.

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

[98] J. Wu, "A Fault-Tolerant and Deadlock-Free Routing Protocol in 2D Meshes Based on Odd-Even Turn Model," *IEEE Trans. Comput.*, vol. 52, pp. 1154–1169, Sept. 2003.

[99] G.-M. Chiu, "The odd-even turn model for adaptive routing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, pp. 729–738, July 2000.

[100] K. Goossens, J. Dielissen, and A. Rădulescu, "ÆThereal network on chip: Concepts, architectures, and implementations," *IEEE Des. Test*, vol. 22, pp. 414–421, Sept. 2005.

[101] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "Hermes: An infrastructure for low area overhead packet-switching networks on chip," *Integr. VLSI J.*, vol. 38, pp. 69–93, Oct. 2004.

[102] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "Qnoc: Qos architecture and design process for network on chip," *J. Syst. Archit.*, vol. 50, pp. 105–128, Feb. 2004.

[103] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "Cadp 2011: a toolbox for the construction and analysis of distributed processes," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 2, pp. 89–107, 2012.

[104] R. J. van Glabbeek, "The linear time-branching time spectrum i - the semantics of concrete, sequential processes," in *Handbook of Process Algebra* (A. P. J.A. Bergstra and S. Smolka, eds.), pp. 3–99, Elsevier, 2001.

[105] R. De Nicola and M. Hennessy, "Testing equivalences for processes," *Theor. Comput. Sci.*, vol. 34, pp. 83–133, 1984.

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

[107] R. J. van Glabbeek, B. Luttik, and N. Trcka, "Branching bisimilarity with explicit divergence," *Fundam. Inform.*, vol. 93, no. 4, pp. 371–392, 2009.

[108] D. Borrione, M. Boubekeur, L. Mounier, M. Renaudin, and A. Sirianni, "Validation of asynchronous circuit specifications using IF/CADP," in *VLSI-SOC: From Systems to Chips, Selected papers from the 12th IFIP International Conference on VLSI*, vol. 200, pp. 85–100, International Federation for Information Processing, December 2006.

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

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

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

[112] V. A. Palaniveloo and A. Sowmya, "Application of formal methods for system-level verification of network on chip," in *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2011, 4-6 July 2011, Chennai, India*, pp. 162–169, IEEE Computer Society, 2011.

[113] L. G. Iugan, G. Nicolescu, and I. O'Connor, "Modeling and formal verification of a passive optical network on chip behavior," *ECEASST*, vol. 21, 2009.

[114] 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, 2011.

[115] F. Verbeek and J. Schmaltz, "A Decision Procedure for Deadlock-Free Routing in Wormhole Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 1935–1944, 2014.

[116] F. Verbeek and J. Schmaltz, "Automatic verification for deadlock in networks-on-chips with adaptive routing and wormhole switching," in *NOCS 2011, Fifth ACM/IEEE International Symposium on Networks-on-Chip, Pittsburgh, Pennsylvania, USA, May 1-4, 2011* [157], pp. 25–32.

[117] A. Alhussien, F. Verbeek, B. van Gastel, N. Bagherzadeh, and J. Schmaltz, "Fully reliable dynamic routing logic for a fault-tolerant noc architecture," *Journal of Integrated Circuits and Systems*, vol. 8, no. 1, pp. 43–53, 2013.

[118] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz, "A formal approach to the verification of networks on chip," *EURASIP Journal Embedded Systems*, vol. 2009, pp. 2:1–2:14, Jan. 2009.

[119] A. Helmy, L. Pierre, and A. Jantsch, "Theorem proving techniques for the formal verification of NoC communications with non-minimal adaptive routing," in *DDECS*, pp. 221–224, IEEE, 2010.

[120] F. Verbeek and J. Schmaltz, "Easy Formal Specification and Validation of Unbounded Networks-on-Chips Architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, pp. 1:1—-1:28, Jan. 2012.

[121] P. Crouzen and F. Lang, *Fundamental Approaches to Software Engineering: 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*, ch. Smart Reduction, pp. 111–126. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[122] M. Gazda and W. Fokkink, "Congruence from the Operator's Point of View: Compositionality Requirements on Process Semantics," in *SOS*, vol. 32 of *EPTCS*, pp. 15–25, 2010.

[123] H. Garavel, W. Serwe, and G. Smeding, "LNT.OPEN manual page."

[124] R. Mateescu and A. Wijs, *Model Checking Software: 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, ch. Property-Dependent Reductions for the Modal Mu-Calculus, pp. 2–19. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[125] R. Mateescu and E. Oudot, "Bisimulator 2.0: An On-the-Fly Equivalence Checker based on Boolean Equation Systems," in *Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2008 (Anaheim, CA, USA)*, pp. 73–74, IEEE Computer Society Press, June 2008.

[126] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, (New York, NY, USA), pp. 110–121, ACM, 2005.

[127] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, *Model Checking Software: 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008 Proceedings*, ch. Efficient Stateful Dynamic Partial Order Reduction, pp. 288–305. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[128] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), pp. 373–384, ACM, 2014.

[129] A. Valmari, "Error Detection by Reduced Reachability Graph Generation," in *Proc. of the 9th European Workshop on Application and Theory of Petri Nets*, (Venice, Italy), 1988.

[130] A. Valmari, *Advances in Petri Nets 1990*, ch. Stubborn sets for reduced state space generation, pp. 491–515. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991.

[131] A. Valmari and H. Hansen, "Can Stubborn Sets Be Optimal?," *Fundam. Inf.*, vol. 113, pp. 377–397, Aug. 2011.

[132] P. Godefroid and D. Pirottin, "Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract)," in *CAV*, pp. 438–449, 1993.

[133] P. Godefroid and P. Wolper, "A Partial Approach to Model Checking," *Inf. Comput.*, vol. 110, no. 2, pp. 305–326, 1994.

[134] P. Godefroid and P. Wolper, *Computer Aided Verification: 3rd International Workshop, CAV '91 Aalborg, Denmark, July 1–4, 1991 Proceedings*, ch. Using partial orders for the efficient verification of deadlock freedom and safety properties, pp. 332–342. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992.

[135] P. Godefroid, G. J. Holzmann, and D. Pirottin, "State-space caching revisited," *Formal Methods in System Design*, vol. 7, no. 3, pp. 227–241.

[136] K. Varpaaniemi, *Application and Theory of Petri Nets 1994: 15th International Conference Zaragoza, Spain, June 20–24, 1994 Proceedings*, ch. On combining the stubborn set method with the sleep set method, pp. 548–567. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.

[137] D. Peled, *Computer Aided Verification: 6th International Conference, CAV '94 Stanford, California, USA, June 21–23, 1994 Proceedings*, ch. Combining partial order reductions with on-the-fly model-checking, pp. 377–390. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.

[138] G. J. Holzmann, P. Godefroid, and D. Pirottin, "Coverage Preserving Reduction Strategies for Reachability Analysis," in *Proceedings of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification XII*, (Amsterdam, The Netherlands), pp. 349–363, North-Holland Publishing Co., 1992.

[139] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening, "Unfolding-based partial order reduction," in *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015* (L. Aceto and D. de Frutos-Escrig, eds.), vol. 42 of *LIPIcs*, pp. 456–469, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[140] A. Mazurkiewicz, *Petri Nets: Applications and Relationships to Other Models of Concurrency: Advances in Petri Nets 1986, Part II Proceedings of an Advanced Course Bad Honnef, 8.–19. September 1986*, ch. Trace theory, pp. 278–324. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987.

[141] T. Kitai, Y. Oguro, T. Yoneda, E. Mercer, and C. Myer, "Partial order reduction for timed circuit verification based on a level oriented model," in *IEICE Transactions*, vol. E86-D, pp. 2601–2611, 2003.

[142] E. Mercer, *Correctness and Reduction in Timed Circuit Analysis*. Ph.D. dissertation, Dept. Elect. & Comp. Eng., Univ. of Utah, Salt Lake City, UT, 2002.

[143] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[144] Y. Zhang, E. Rodriguez, H. Zheng, and C. Myers, "An Improvement in Partial Order Reduction Using Behavioral Analysis," in *IEEE Computer Society Annual Symposium*, 2012.

[145] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. Ph.D. dissertation, Sch. Comp. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, 1988.

[146] A. J. Martin, "Self-timed fifo: An excercise in compiling programs into vlsi circuits," tech. rep., California Institute of Technology, 1986.

[147] P. Wolper and D. Leroy, *Computer Aided Verification: 5th International Conference, CAV '93 Elounda, Greece, June 28–July 1, 1993 Proceedings*, ch. Reliable hashing without collision detection, pp. 59–70. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993.

[148] G. J. Holzmann, "Tracing Protocols," *AT&T Technical Journal*, vol. 64, pp. 2413–2434, 1987.

[149] H. R. Andersen, "Partial model checking," in *In Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pp. 398–407, IEEE Computer Society Press, 1995.

[150] F. Lang and R. Mateescu, "Partial model checking using networks of labelled transition systems and boolean equation systems," *Logical Methods in Computer Science*, vol. 9, no. 4, pp. 1–32, 2013.

[151] H. Zheng, "Compositional reachability analysis for efficient modular verification of asynchronous designs," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 29, pp. 329–340, Mar. 2010.

[152] H. Zheng, E. Rodriguez, Y. Zhang, and C. J. Myers, "A Compositional Minimization Approach for Large Asynchronous Design Verification," in *SPIN*, pp. 62–79, 2012.

[153] B. Bartley, J. Beal, C. Kevin, M. Goksel, N. Roehner, E. Oberortner, M. Pocock, M. Bissell, C. Madsen, T. Nguyen, Z. Zhang, J. H. Gennari, C. Myers, A. Wipat, and H. Sauro, "Synthetic biology open language (sbol) version 2.0.0," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, p. 272, 2015.

[154] N. Roehner, Z. Zhang, T. Nguyen, and C. J. Myers, "Generating systems biology markup language models from the synthetic biology open language," *ACS Synthetic Biology*, vol. 4, no. 8, pp. 873–879, 2015. PMID: 25822671.

[155] Z. Zhang, T. Nguyen, N. Roehner, G. Misirli, M. Pocock, E. Oberortner, J. Beal, K. Clancy, A. Wipat, and C. Myers, "libSBOLj 2.0: A Java library to support sbol 2.0." IEEE Life Sciences, in press.

[156] N. Roehner, J. Beal, K. Clancy, B. Bartley, R. Grunberg, G. Misirli, E. Oberortner, M. Pocock, M. Bissell, C. Madsen, T. Nguyen, Z. Zhang, J. H. Gennari, A. Wipat, H. Sauro, and C. J. Myers, "Synthetic biology open language 2.0: Sharing structure and function in biological design." submitted to ACS Synthetic Biology.

[157] *NOCS 2011, Fifth ACM/IEEE International Symposium on Networks-on-Chip, Pittsburgh, Pennsylvania, USA, May 1-4, 2011*, IEEE Computer Society, 2011.