Title
Scalable Model Checking Beyond Safety - A Communication Fabric Perspective

Author
Ray, Sayak

Publication Date
2013

Peer reviewed|Thesis/dissertation

Scalable Model Checking Beyond Safety
A Communication Fabric Perspective

by

Sayak Ray

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Robert K. Brayton, Chair
Professor Sanjit A. Seshia
Professor Lauren K. Williams

Fall 2013

Scalable Model Checking Beyond Safety
A Communication Fabric Perspective

Abstract

Scalable Model Checking Beyond Safety
A Communication Fabric Perspective

by

Sayak Ray

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Robert K. Brayton, Chair

In this research, we have developed symbolic algorithms and their open-source implementations that effectively solve liveness verification problem for industrially relevant hardware systems. In principle, our tool-suite works on any sequential hardware circuit and for the whole family of ω-regular properties. Practicality and effectiveness of our tool-suite have been demonstrated in the context of proving response properties (a very common and important liveness property) of on-chip communication fabrics.

To Maa

Contents

Contents ii

List of Figures iv

List of Tables vi

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Contributions . . . 3
- 1.3 Organization . . . 4

2 Algorithmics for Liveness 6
- 2.1 Safety vs. Liveness . . . 6
- 2.2 LTL Model Checking . . . 6
- 2.3 Algorithms for Liveness Verification . . . 9

3 Formal Model for Communication Fabrics 12
- 3.1 Executable Micro-Architecture Specification (xMAS) . . . 13
- 3.2 Benchmarks . . . 17
- 3.3 xMAS in Perspective . . . 22

4 Bug Hunting for Liveness 26
- 4.1 Introduction . . . 26
- 4.2 Preliminaries . . . 26
- 4.3 L2S Conversion for Stabilization Properties . . . 28
- 4.4 Experimental Results . . . 30

5 Efficient Proof Of Liveness 33
- 5.1 Introduction . . . 33
- 5.2 Credit Mechanism and Buffer Relations . . . 34
- 5.3 Response Formulation . . . 36
- 5.4 Approach I : Breaking Into Safety Properties . . . 37

5.5 Approach II: Well-founded Induction 44
5.6 Approach III : Skeleton Independent Proof Heuristics 55
5.7 Approach IV : Proof based on k-LIVENESS 66

6 Structural Invariant Generation **80**
6.1 Introduction 80
6.2 Related Works 82
6.3 Preliminaries 83
6.4 Invariant Mining For xMAS fabrics 86
6.5 Experimental Results 92
6.6 Conclusion 95

7 Conclusion and Future Work **96**
7.1 Conclusion 96
7.2 Future Work 96

Bibliography **98**

List of Figures

2.1 Taxonomy of SCC analysis algorithms 10

3.1 xMAS symbols for structural components of communication fabrics 13
3.2 A synchronous FIFO queue with a write port (on the left) and a read port (on the right). The queue can store k data elements. In each clock cycle, if the queue is not full, a new element may be inserted; and if the queue is not empty, the oldest element may be removed. read_data exposes the oldest element if the queue is not empty. If the queue is empty, the incoming data appears at the output one cycle later. 14
3.3 Back-to-back composition of two buffers with modified interface signals 14
3.4 A succinct representation of two buffers connected back-to-back from Figure 3.3 . 15
3.5 Credit mechanism 18
3.6 Virtual channel ($\mathcal{VC}$) 19
3.7 Buffered virtual channel ($\mathcal{VCB}$) 20
3.8 Virtual channel with ordering ($\mathcal{VCO}$) 20
3.9 A pair of agents communicating over a simple fabric ($\mathcal{MS}$) 21
3.10 A two entry scoreboard ($\mathcal{SB}$) 22

4.1 Liveness-to-safety transformation for $\mathbf{F}p$ 29
4.2 L2S for stabilization property $FGa \Rightarrow FGb + FGc$ 31

5.1 Credit mechanism 35
5.2 Virtual channel 36
5.3 A deadlocking virtual channel 41
5.4 Well-founded structure for virtual channel 45
5.5 WFS for virtual channel with order 48
5.6 A folded representation the same 49
5.7 Various configurations of B_5 (captured in predicates π_1, π_1, π_2) 50
5.8 Well-founded structure for Master/Slave design 53
5.9 Compositional structure for Master/Slave design 54
5.10 Schematic representation of a linearly graded pending graph 58

5.11 Schematic representation of a topologically graded pending graph 59
5.12 Discovering linear gradation using Boolean signals 61
5.13 Abstract state-space for virtual channel 64
5.14 Abstract state-space (some transitions pruned) for virtual channel 65
5.15 A typical waveform for a signal p that satisfies property $\mathbf{FG}p$ 66
5.16 Absorber Logic that absorbs one 'drop' on p 67
5.17 A typical waveform for signal p that enters the absorber logic of Figure 5.16 and the corresponding waveform for p_{out} 68
5.18 Cascade of absorber logic for absorbing 2 'drops' on p 68
5.19 Waveform for p and p_{out} obtained from a two-level cascade of absorber circuit . . 69
5.20 Formation of arena with stabilizing constraints and their role in precluding unreachable loops 70

6.1 Credit mechanism 81
6.2 A directed graph and one of its spanning tree 84
6.3 $\mathcal{VCO}$: A virtual channel with ordering 88
6.4 Sub-network that steers $type(A_1)$-flit 88
6.5 Sub-network that steers $type(A_2)$-flit 89
6.6 Sub-network of Figure 6.4 simplified by 'shorting' arbiters and switches 89
6.7 Sub-network of Figure 6.5 simplified by 'shorting' arbiters and switches 90
6.8 Marked graphs for the sub-networks of $\mathcal{VCO}$ 91

List of Tables

4.1 $\mathcal{CL}$ 32
4.2 $\mathcal{VC}$ 32
4.3 $\mathcal{MS}$ 32

5.1 Auxiliary safety assertions 39
5.2 Assumptions used in the proofs 42
5.3 Experiment on various communication fabrics 43
5.4 Verification run-times for intermediate assertions on well-foundedness 55
5.5 Verification run-times for skeleton independent proof 65
5.6 Liveness property verification run-time with liveness-to-safety transformation . . . 76
5.7 Performance of k-LIVENESS with arenaViolation 77
5.8 Performance of k-LIVENESS without arenaViolation 77

6.1 Correspondence between vertices of marked graph in Figure 6.8(a) and xMAS components of Figure 6.6 90
6.2 Correspondence between vertices of marked graph in Figure 6.8(b) and xMAS components of Figure 6.7 90
6.3 Edge identifiers for the marked graphs 91
6.4 Assumptions used in the proofs 93
6.5 Experiment on various communication fabrics (1) 94
6.6 Experiment on various communication fabrics (2) 94

Acknowledgments

I thank Prof. Robert Brayton for advising me during my graduate study. His open and positive outlook has remained the principal guideline in my research. I hope this will continue to steer my way from now on. He will always remain a colossal figure of inspiration to me. Alan Mishchenko taught me the basic skills of research from the ground up. He showed me what it means to be passionate about research. He was my guide in almost every path that I explored in the past few years, be it an intricate research idea of AIG isomorphism or a fun hike in the Columbia River Gorge. I found him relentlessly enthusiastic whenever I approached him with a problem. I was fortunate to have Niklas Een, Baruch Sterin and Jiang Long as my colleagues in Bob's group. Working with them was an enriching experience. I thank to Mike Case for always responding to my queries and concerns.

Satrajit Chatterjee has deep influence in this dissertation. We had long hours of research discussions in Intel, and Satrajit made them productive and joyful with his own charisma. Some of the works presented in this dissertation began under his mentorship. I am indebted to him for being a caring mentor in my research projects and beyond. I thank Michael Kishinevsky for letting me and Satrajit work together under his watchful eyes. Timothy Kam, Qi Zhu, Emily Shriver, Umit Ogras, Murali Talupur and the full team of Strategic CAD Lab made my Intel internships fruitful and memorable during my formative years. I thank Prosenjit Chatterjee, Yogesh Mahajan and Vigyan Singhal for arranging another wonderful internship in Nvidia.

I thank Prof. Sanjit Seshia and Prof. Lauren Williams for being members of my dissertation committee. Prof. Seshia's comments and feedback on the initial draft of the dissertation were very helpful. I acknowledge Prof. Edward Lee for his feedback on my qualifier examination. I am deeply grateful to Prof. Koushik Sen and Prof. Jaijeet Roychowdhury for being my temporary advisers and funding my research efforts during the initial and final months of my study respectively. Stavros Tripakis was a great peer in the DOP Center with whom I had many stimulating research discussions. I gratefully acknowledge Prof. Christos Papadimitriou's support in the Fall 2010 semester. Shirley Salanio and Ruth Gjerde were tirelessly supportive in all of my graduate matters. This dissertation would not have been possible without this collaborative Cal community.

I thank Prof. P. P. Chakrabarti, Prof. Pallab Dasgupta and Prof. Dipankar Sarkar for motivating and preparing me for the graduate study. Without their inspiration and support, I could not have experienced the wonders of Berkeley. I sincerely acknowledge Prof. Goutam Biswas, Prof. Abhijit Das, Prof. Arijit Bishnu, Prof. Arobinda Gupta and Prof. Chandan Mazumdar for their encouragements.

I enjoyed few great years in the DOP Center with Mehdi Maasoumy, Susmit Jha, Rhishikesh Limaye, Daniel Holcomb, Wenchao Li, Dai Bui, Tobias Welp and Pierluigi Nuzzo. I had brief but quite memorable overlap with Xuening Sun, Shauki Elassaad, Aaron Hurst, Zile Wei, Shanna-Shaye Forbes, Bryan Brady, Alberto Puggelli, Antonio Iannopollo, Yen-Sheng Ho, Aaditya V. Karthik and Rohit Sinha. I thank James Cook, Anindya De, Kurt Miller and Siu

Man Chan for being my incredibly helpful GSI colleagues. My transition from India to United States would never have been a smooth one without Avishek Saha and Anirvan Duttagupta's help. I appreciate Partha Sarathi De, Arnab Sen, Subhroshekhar Ghosh, Koushik Pal, Sharmodeep Bhattacharya, Riddhipratim Basu, Sujayam Saha and Shubhankar Chaudhuri for creating a homely space around me that I left back in my home-town.

I thank the members of my family for their endless good wishes which have been instrumental to all my decisions and reflections. Their unconditional support during my times of difficulties can hardly be reciprocated.

Chapter 1

Introduction

This dissertation belongs to the broad area of *formal methods for system design*. Its overarching theme is to develop scalable algorithms and tools for *liveness verification*. Liveness is a type of *formal specification* used to specify certain behaviors of hardware and software systems. The other type, which is relatively simpler than liveness, is called *safety*. Safety properties specify that something bad will never happen. For example, the property of mutual exclusion, which specifies that two concurrent processes will never access a critical section simultaneously, is a safety property. Liveness properties, on the other hand, specify that something good will happen eventually. Scenarios like deadlock-freedom, livelock-freedom, stabilization etc. are modeled using liveness properties. In our research, we have developed various symbolic algorithms and their open-source implementations that are effective in solving liveness verification problems for industrially relevant hardware systems and outperform traditional algorithms. In principle, the tool-suite that we have developed in this thesis works on any sequential hardware circuit, and for the whole family of *ω-regular properties* [Thomas, 1990]. In particular, practicality and effectiveness of our tool-suite have been demonstrated in the context of proving *response properties*[1] of on-chip communication fabrics. This tool-suite consists of a proof engine and a bug-hunting engine. The proof engine extends the recent development of *k*-LIVENESS proposed by Claessen and Sörensson [Claessen and Sörensson, 2012]. The bug-finder is based on the liveness-to-safety conversion algorithm originally proposed by Biere *et al.* [Schuppan and Biere, 2004]. Following are the three main components of this dissertation:

Proof Engine The main thrust of our proof engine is scalability. In order to achieve this, we have deviated from the classic approaches like *nested fix-point formulation* or *strongly connected component enumeration* for reasoning about liveness, and resorted to a more scalable technique called *ranking oriented proof*. In general, automatically finding a ranking oriented proof is known to be a challenging task; there is no known mathematical theory for extracting ranking information from an arbitrary finite state machine

[1] A response property is a common and important liveness property. See Section 5.3 for further details.

represented as a sequential circuit. To address this challenge, we devised a specialized *constraint mining algorithm* that discovers special kinds of Boolean relations from the target hardware. These relations can convey useful ranking information for various types of hardware of practical importance. In fact, these ranking constraints played an instrumental role in the scalability of our tool-suite for proving response properties of communication fabrics.

Bug Finder The core idea of k-LIVENESS is aimed at *proving* a liveness property, but it cannot disprove an incorrect property *i.e.* it cannot find a bug if one exists. Thus to complete our tool-suite, a separate bug-finder was developed. In principle, it can both prove or disprove a liveness property, but its underlying algorithm is more effective for finding bugs using bounded model checking or random simulation.

Analysis of Invariants In addition to advancing core techniques for liveness verification, this dissertation also emphasizes the *impact of design invariants* on scalable liveness verification. It is well-known that design invariants can be crucial ingredients for scalable safety verification. Because we leverage safety verification technology for liveness verification in our work, we demonstrate that the role of design invariants can be extended naturally to liveness verification. With the introduction of design invariants in our experiments, we achieve a speed-up in liveness verification for our target communication fabrics (see experimental results in Chapter 5). For these systems, Chatterjee and Kishinevsky presented an algorithm [Chatterjee and Kishinevsky, 2010a] that discovers a set of linear invariants automatically. These invariants were shown to be crucial for efficient safety verification. Our experiments show that these invariants are essential for efficient liveness verification as well. Additionally, we provide a novel and rigorous mathematical justification about the scope of the Chatterjee-Kishinevsky algorithm. We show that the classic notion of *Kirchhoff's voltage law* (more fundamentally, the null-space analysis of the incidence matrix of a directed graph) offers a precise mathematical tool for mining this particular kind of invariants from communication fabrics.

1.1 Motivation

Theoretical foundations for safety and liveness verification (*eg.* theory of ω-regular model checking) have been developed by researchers over the last four decades. Comprehensive books like [Clarke *et al.*, 1999] and [Baier and Katoen, 2008] are available as authoritative expositions on this topic. Based on this rich body of foundational knowledge, significant effort is now being invested, both in academia and industry, in transforming this theory into useful software technology that engineers can use to solve their verification problems arising in industry. Safety verification, being the first hurdle in this direction, has received the lion's share of attention from researchers so far. As a result, today there exist extremely sophisticated symbolic algorithms for reachability analysis and their heavily engineered implementations that can solve many industrially important safety verification problems. These were beyond the capacity of available technologies even five years ago. Liveness verification, being the second hurdle in this area, is yet to receive a comparable effort. This dissertation is a step in bridging this gap. It aims to leverage knowledge and techniques acquired from the many advances in safety verification and to develop successful technologies for liveness verification.

1.2 Contributions

This dissertation offers new capabilities of bit-level liveness verification in general and an in-depth formal analysis of communication fabrics with the objective of their scalable bit-level response verification. At the conceptual level, we make the following contributions:

Well-foundedness in Hardware Systems: We show that the operations of industrially relevant hardware systems (communication fabrics in our case studies) can naturally give rise to a *well-founded ordering* in their state spaces. We call such well-founded ordering a *ranking structure*. We show how we can use *disjunctive stabilizing assertions* to capture this ranking structure, and subsequently use them to speed up liveness verification. We propose an algorithm that discovers these stabilizing assertions automatically from the bit-level netlists. We demonstrate a few other heuristic techniques that can leverage these ranking structures implicitly to speed up liveness verification. These heuristic techniques improve on verification runtime at the expense of lesser automation.

Invariant Characterization: We provide an algebraic analysis of a set of industrially relevant communication fabrics. The analysis offers a rigorous mathematical justification of the Chatterjee-Kishinevsky invariants and their connection to the underlying network topology. Our analysis enhances our understanding about the Chatterjee-Kishinevsky invariants which are crucial for timely convergence of verification experiments on the target fabrics.

This dissertation involves significant experimental work. We have supported our conceptual claims and contributions with numerous experiments and benchmarking. Various prototypical tools have been developed and released as part of this research as described below:

Open Source Tool and Benchmark Development: Our proof and debug engines for liveness are developed in the ABC verification environment [Mishchenko, 2013]. ABC is an open-source hardware synthesis and verification tool that offers state-of-the-art safety verification capabilities. Mainly utilizing ABC's netlist manipulation functions and property directed reachability engine, we have developed our liveness verification engines which are now publicly available with the official distribution of ABC. The now available ABC commands ***kcs***, ***l2s*** and ***l3s*** correspond to our proof and debug engines. While command ***kcs*** offers the prototypical implementation of the k-LIVENESS algorithm extended with the notion of disjunctive stabilizing constraints, commands ***l2s*** and ***l3s*** offer two variants of liveness-to-safety conversion algorithm for stabilization properties. In order to perform verification experiments, we have developed our own implementation of component libraries for communication fabrics (see Chapter 3 for details). This includes Verilog implementation of libraries and a code generator for fabrics from high-level textual description of fabric connections. We have developed various benchmarks of industrially relevant, publicly available fabric designs using our framework. These benchmarks have been used extensively in all our experiments and are publicly available for further experimentation.

1.3 Organization

The remaining chapters of this dissertation are organized as follows:

Chapter 2, 3 : Background Concepts

We recapitulate necessary background concepts in Chapter 2 and 3. Chapter 2 discusses the basics of model checking for *linear temporal logic* (LTL) properties and the traditional algorithms for solving the liveness verification problem. Chapter 3 discusses the xMAS framework. xMAS (Executable Micro-Architecture Specification) is a high-level modeling framework suitable for design and formal analysis of communication fabrics. It was proposed by researchers at the Intel Strategic CAD Laboratory [Chatterjee *et al.*, 2010] and has gained popularity among practitioners in very short time. After describing the basic library primitives of xMAS, Chapter 3 discusses xMAS models for a collection of industrially relevant fabrics in detail. These models are used as benchmarks in our experiments.

Chapter 4 : The Bug Finder

Our liveness-to-safety bug-hunting technique is described in Chapter 4. It first introduces *stabilization properties*, a syntactic subset of LTL which is semantically equivalent to the entire class of ω-regular properties. Then it describes a simple construction for converting a stabilization verification problem into an equi-satisfiable safety verification problem. This construction enables bounded model checking and random simulation to be used for bug hunting for stabilization properties.

Chapter 5 : Proof Techniques

Chapter 5 is devoted to *proof techniques* for liveness. While our main objective is to discuss our liveness proof engine that operates on the k-LIVENESS principle, we gradually build the idea by illustrating how ranking structures are hidden in the state spaces of communication fabrics and how our techniques discover and leverage them. First we explain how the end-to-end response property can be split into intermediate safety properties (Chapter 5.4). As proof obligations, these intermediate safety properties are easier than the end-to-end liveness one. This makes the scheme an attractive way to cope with the state explosion problem associated with liveness verification. However, this scheme requires significant manual intervention; a rigorous understanding of the design is required because there exists no automatic method for splitting an end-to-end liveness property into intermediate safety properties. Also, one needs to argue that satisfaction of the intermediate safety properties implies satisfaction of the liveness property. The nature of this argument is situation-specific and usually ad-hoc. One way of mitigating this issue is to use the notion of *well-founded ordering*. We demonstrate how this well-founded ordering information can be extracted from the design description of communication fabrics (Chapter 5.5). While the notion of well-founded ordering offers a systematic mathematical reasoning about liveness (contrary to the ad-hoc reasoning involved in the technique of intermediate safety properties), still the overall process is manual. In general, we do not know any algorithmic way of discovering a well-founded ordering from a design description. For our benchmarks, the ordering information is discovered manually. We increase the level of automation in the subsequent section (Chapter 5.6) by using a heuristic proof technique called *skeleton independent proof* proposed by Aaron Bradley *et al.* [Bradley *et al.*, 2011]. Finally, we discuss our algorithmic proof technique based on k-LIVENESS at the end (Chapter 5.7). In principle, although it is a fully automatic algorithm, it can benefit from human guidance as well. A comparison among all these alternative techniques for proving liveness is discussed in detail in Chapter 5.

Chapter 6 : Structural Invariant Generation

Our contribution to invariant generation for communication fabrics is discussed in Chapter 6. We summarize some necessary background on linear algebra and related works at the beginning of the chapter. Then we demonstrate how Kirchhoff's voltage law can be leveraged to generate invariants for communication fabrics.

Chapter 7 : Conclusion

Finally, Chapter 7 concludes the dissertation with a discussion on interesting possibilities of future investigations.

Chapter 2

Algorithmics for Liveness

2.1 Safety vs. Liveness

Safety and liveness are the main categories of temporal specification for reactive systems. Informally, safety properties specify that something 'bad' will never happen and liveness properties specify that something 'good' will happen eventually. The notions of safety and liveness was introduced first by Leslie Lamport in the 1970s in the context of the analysis of distributed systems [Lamport, 1980]. After this, numerous specification formalisms were proposed to capture these requirements in rigorous mathematical frameworks. These formalisms include *linear temporal logic* (LTL), *computation tree logic* (CTL), *ω-regular specification* etc. In this dissertation, we adopt LTL for specification since it is the most widely used temporal logic and particularly appealing for liveness specification. We devote this chapter to outline syntax, semantics and model checking algorithms for LTL. Since an extensive literature is available on this topic, we focus only on the aspects that are most relevant to our work i.e. the algorithms for liveness verification.

The chapter is organized in two sections: Section 2.2 overviews the basics of LTL model checking and Section 2.3 discusses the taxonomy of the most prominent liveness verification algorithms.

2.2 LTL Model Checking

LTL is an extensively studied temporal logic and used widely for specifying behavior of reactive systems. LTL model checking is now an entry-level topic in formal methods with a plethora of literature available on specification and verification of LTL formulas. For completeness, we present a brief overview of syntax, semantics and basic model checking algorithm for LTL here. It is only a quick outline and does not attempt to reveal the full technical depth of the topic. We refer curious readers to comprehensive books like [Clarke *et al.*, 1999], [Baier and Katoen, 2008] for full details.

2.2.1 Syntax of LTL

Let AP be a set of atomic propositions and let $p \in AP$ be some atomic proposition. LTL has the following syntax given in Backus-Naur form:

$$\phi := \top \quad | \quad \bot \quad | \quad p \quad | \quad \neg\phi \quad | \quad \phi \wedge \phi \quad | \quad \mathbf{X}\phi \quad | \quad \phi\mathbf{U}\phi$$

Here symbols $\top$ and $\bot$ stand for logical constants **true** and **false** respectively. $\neg$ and $\wedge$ are standard Boolean connectives denoting negation and conjunction respectively. **X** and **U** are temporal connectives that stand for **next** and **until** respectively. The above syntax is minimalistic in that it avoids many other (standard) Boolean and temporal connectives that can be expressed with the above connectives. For example, often we will use Boolean expressions $\phi_1 \vee \phi_2$ (disjunction) and $\phi_1 \Rightarrow \phi_2$ (implication) in place of $\neg(\neg\phi_1 \wedge \neg\phi_2)$ and $\neg\phi_1 \vee \phi_2$ respectively. Similarly, $\mathbf{F}p$ and $\mathbf{G}p$ are standard LTL expressions that are shorthands of $\top\mathbf{U}p$, and $\neg(\top\mathbf{U}\neg p)$ respectively. **F** and **G** stand for **future** and **globally** respectively.

2.2.2 Semantics of LTL

The semantics of LTL is defined over a formal model of transition systems called *Kripke structure*. A standard definition of Kripke Structure is given below:

Definition 1 (Kripke Structure)**.** A Kripke structure $K = (S, S_0, R, L)$ is defined over a set of atomic propositions AP such that S is a set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a *total* transition relation and $L : S \rightarrow 2^{AP}$ is a labelling function.

A *path* π in a Kripke structure K is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$ for all possible $i \geq 0$ and $s_0 \in S_0$. In our discussion, we consider only infinite paths in a Kripke structure unless stated otherwise. The i-th state on a path π is denoted as $\pi(i)$, for $i \geq 0$ and the suffix sub-path of π that starts from $\pi(i)$ is denoted as π^i. Using symbol $\models$ to denote logical satisfaction relation, the semantics of LTL is defined as follows:

- $K, \pi \models p \Leftrightarrow p \in L(\pi^0)$
- $K, \pi \models \neg\phi \Leftrightarrow K, \pi \not\models \phi$
- $K, \pi \models \phi_1 \wedge \phi_2 \Leftrightarrow K, \pi \models \phi_1 \wedge K, \pi \models \phi_2$
- $K, \pi \models \mathbf{X}\phi \Leftrightarrow K, \pi^1 \models \phi$
- $K, \pi \models \phi_1\mathbf{U}\phi_2 \Leftrightarrow$ there exists $j \geq 0$ such that $K, \pi^j \models \phi_2$ and $K, \pi^i \models \phi_1$ for all $i < j$.

2.2.3 Model Checking Algorithm for LTL

We overview the basic LTL model checking algorithm based on *Büchi automaton* construction below. We first summarize the model checking algorithm, then recapitulate the notion of Büchi automaton.

2.2.3.1 The Basic LTL Model Checking Algorithm

Let K be a Kripke structure representation of some reactive system. Let ϕ be an LTL formula over a set of atomic propositions AP that specifies all bad behaviors. We want to prove that $K \models \neg\phi$ *i.e.* K does not admit any bad behavior specified by ϕ. We build a *Büchi automaton* A_ϕ that accepts all sequences over 2^{AP} satisfying ϕ. We then check whether $\mathcal{L}(A_\phi) \cap \mathcal{L}(K) = \emptyset$ where $\mathcal{L}(X)$ denotes the language accepted by automaton X. If $\mathcal{L}(A_\phi) \cap \mathcal{L}(K) = \emptyset$, then K satisfies $\neg\phi$; otherwise we obtain a *counterexample*. Model checking of LTL is a PSPACE-complete problem in general. For details of the model checking procedure see [Clarke *et al.*, 1999], [Vardi and Wolper, 1984].

2.2.3.2 Büchi automaton

A Büchi automaton is a type of automaton on infinite strings and lies at the heart of LTL model checking. It has been studied extensively by automata theorists and logicians. Although in this dissertation we never construct a Büchi automaton explicitly, still it is an important background concept, particularly for Chapters 4 and 5. For completeness, we review the definition of a Büchi automaton and algorithm for its emptiness check. For further details see [Thomas, 1990]. Any LTL property can be translated into a Büchi automaton, but we omit details of this construction here and refer interested readers to [Kupferman and Vardi, 2005].

Definition 2 (Büchi Automaton)**.** A Büchi automaton is defined on an alphabet Σ as a quadruple $\langle S, S_0, R, F\rangle$ where S is a (finite) set of states, $S_0 \subseteq S$ is the set of initial state, $R \subseteq S \times \Sigma \times S$ is a set of transitions (labeled with symbols from Σ) and $F \subseteq S$ is the set of accepting states.

Let Σ^ω denote the set of all infinite strings of symbols from Σ. A Büchi automaton accepts strings from Σ^ω. For a string $a \in \Sigma^\omega$ where $a = a_0, a_1, a_2, \ldots$ for $a_i \in \Sigma$, a *run* of a on a Büchi automaton $B = \langle S, S_0, R, F\rangle$ is an infinite sequence of states $\rho_a = s_0, s_1, s_2, \ldots$ such that all $s_i \in S$, $s_0 \in S_0$ and $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. Let $Inf(\rho_a) \subseteq S$ denote the set of states in the run ρ_a that appear infinitely many times. Now, a run ρ_a is an *accepting run* if $Inf(\rho_a) \cap F \neq \emptyset$. A string $a \in \Sigma^\omega$ is accepted by Büchi automaton B if it produces an accepting run. The set of all infinite strings accepted by B is called the *language* of B and is denoted by $\mathcal{L}(B)$.

2.2.4 Emptiness check of Büchi Automaton

The problem of emptiness check of a Büchi automaton is a fundamental concern in formal verification. It is not difficult to see that the product automaton $A_\phi \times K$ is also a Büchi automaton and hence the model checking question for LTL *i.e.* if $\mathcal{L}(A_\phi) \cap \mathcal{L}(K) = \emptyset$ reduces to the emptiness check of a Büchi automaton. Given a Büchi automaton $B = \langle S, S_0, R, F \rangle$, $\mathcal{L}(B)$ is empty if and only if B can not produce an infinite sequence of states obeying R starting from an initial state that visits at least one state in F infinitely often. Since the set of states S of B is finite, an infinite sequence of states that obeys R must form a *lasso loop*, a run of states that starts from an initial state and eventually enters and remains in a cycle of states. Moreover, if such an infinite sequence must visit at least one state in F infinitely many times, the loop part of the lasso loop must have at least one state in F. Therefore, a necessary and sufficient condition for $\mathcal{L}(B) = \emptyset$ is that the underlying graph of B cannot have a cycle, reachable from an initial state, that contains a state in F. In other words, no reachable *maximal strongly connected component* (MSCC) of B can contain a state in F. An algorithm for finding strongly connected component (SCC) in a directed graph can, therefore, solve the emptiness checking problem of a Büchi automaton as well as the LTL model checking problem, at least in theory.

2.3 Algorithms for Liveness Verification

At the heart of any LTL liveness verification algorithm, there is a variant of SCC analysis routine. Over the last three decades, researchers came up with different techniques of SCC analysis to improve the scalability of LTL model checking. Early SCC analysis algorithms date back to Tarjan's work on depth first search and related linear-time graph traversal algorithms [Tarjan, 1972]. These algorithms, though have linear time-complexity, rely on explicit state traversal which makes them un-usable for model checking because explicit construction of graphs of transitions systems is infeasible in practice. Therefore, researchers proposed algorithms that perform state traversal implicitly a.k.a. *symbolically*. Thus there exist two broad categories of SCC analysis algorithms, *explicit state traversal* algorithms (*i.e.* the Tarjan-style classical algorithms) and *symbolic state traversal* algorithms. Because the explicit algorithms are rarely used for practical model checking, we omit further discussion about them and refer to algorithm text-books like [Cormen *et al.*, 2009] for their description. Instead, we focus on the family of symbolic algorithms that are actively used for model checking in practice. The taxonomy of different SCC analysis algorithms is presented in Figure 2.1.

As shown in Figure 2.1, the class of symbolic algorithms is divided into *BDD-based algorithms* and *SAT-based algorithms*. *BDD* is the acronym for *binary decision diagram* [Bryant, 1992] - a data structure used for canonical representation of Boolean functions. *SAT*, on the other hand, stands for Boolean satisfiability solver [Eén and Sörensson, 2003].

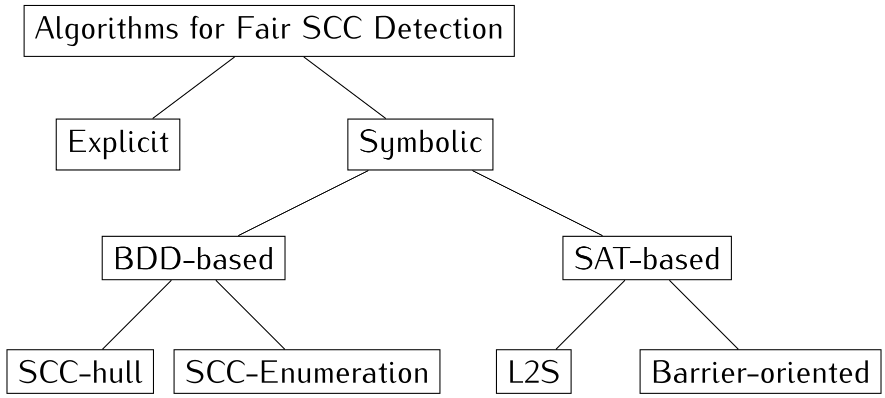


Figure 2.1: Taxonomy of SCC analysis algorithms

Under BDD-based algorithms, there are *SCC hull-finding algorithms* and *SCC enumeration algorithms*. SAT-based algorithms are split into *liveness-to-safety conversion algorithms* (L2S) and *barrier-oriented algorithms*. Before describing them in detail, we make a remark on the nomenclature of the algorithms:

Remark 1. we use the terms 'BDD-based' and 'SAT-based' in order to emphasize the underlying verification technologies that are used with the corresponding algorithms. This attribute is not necessarily stringent because the algorithms that are designated as SAT-based can be solved with BDDs as well (if tractable). However, the formulations discussed under the BDD-based category are generally not SAT-solver friendly. A stronger distinguishing feature for these algorithms is the time period of their development. All algorithms discussed in the BDD-based category were developed in the late 1980s and the 1990s. SAT-based algorithms came after 2000 as SAT solvers began to outperform BDDs in solving large practical problems, thanks to successfully engineered SAT solvers like zCHAFF [Zhang and Malik, 2002] and MiniSat [Eén and Sörensson, 2003]. Liveness-to-safety conversion was proposed around 2002 mainly as a novel theoretical concept, without any special attachment to the use of SAT-solvers. Effective use of SAT-solvers with this formulation is an added plus. An alternative theme that may be identified among the SAT-based symbolic SCC analysis algorithms is '*looking at the liveness verification problem from a safety verification standpoint*'. We expand more on this as we present the algorithms below.

2.3.1 BDD-based algorithms

BDD-based algorithms may be divided into two categories, viz. SCC hull-finding algorithms and SCC enumeration algorithms. Among the SCC hull-finding algorithms, the best-known is the Emerson-Lei algorithm [Emerson and Lei, 1986]. It works on the nested fixpoint formulation for finding MSCC hulls. Several variants have been proposed, for example see [Hojati *et al.*, 1993a], [Hojati *et al.*, 1993b], [Kesten *et al.*, 1998], [Hardin *et al.*, 2001]. These variants achieve better speed-ups on certain types of state graphs. [Somenzi *et al.*, 2002] provides a comprehensive comparison and a generalization of these ideas. [Xie and Beerel, 1999] proposed a BDD-based algorithm for explicit enumeration of the SCCs and improvements on this idea were proposed later in [Bloem *et al.*, 2006]. However, these BDD-based algorithms, though elegant, do not scale in practice. [Ravi *et al.*, 2000] provides a survey, and a comparative analysis of these algorithms.

2.3.2 SAT-based algorithms

As mentioned before, this family of algorithms works on formulations that are solvable by both BDD and SAT-solvers. But their ability of working with SAT-solvers makes them more attractive in practice. These algorithms have a deep connection to modern verification techniques for safety properties and their developments were inspired by the practical success of contemporary safety verification technologies. The liveness-to-safety conversion technique is such a method that converts a liveness problem into an equi-satisfiable safety problem [Schuppan and Biere, 2004]. This technique applies to LTL formulas as well as ω-regular properties. This technique forms the core of our work on a BMC-based debugger for stabilization properties presented in Chapter 4. We defer an in-depth discussion of liveness-to-safety conversion until that chapter.

In the last few years, significant development has taken place in SAT-based algorithms for liveness verification. A liveness track was introduced in the hardware model checking competition and the top contending tools offered advanced SAT-based algorithms for liveness, the top two being IIMC and TIP. The algorithms behind these tools are known as FAIR [Bradley *et al.*, 2011] and k-LIVENESS [Claessen and Sörensson, 2012] respectively. Algorithm FAIR introduced the idea of *barriers* in the context of symbolic SCC analysis. Barriers are inductive invariants that separate MSCCs of a directed graph. FAIR proves a liveness property by iteratively discovering barriers, thereby discovering SCC hulls in a symbolic and incremental way. Bradley *et al.* introduced a heuristic called *skeleton independent proof* in [Bradley *et al.*, 2011] which forms the foundation of Chapter 5.6 of this dissertation. k-LIVENESS, which was the winner in the liveness track of the recent-most hardware model checking competition [HWM, 2012] borrows the idea of barriers in a subtle way and offers a more scalable algorithm. This is the reason for putting these two algorithms together in the barrier-oriented category in (Figure 2.1). k-LIVENESS is the core of our Chapter 5.7. We will discuss these two algorithms in further detail in Chapter 5. In the particular context of liveness verification algorithms for communication fabrics, Holcomb *et al.* proposed a SAT-based algorithm for bounded liveness and performance analysis [Holcomb *et al.*, 2012].

Chapter 3

Formal Model for Communication Fabrics

Communication fabrics are integral part of contemporary hardware systems. They are modules of logic that connect various computation units and I/O units within a large system and mediate data communication among them. The term 'communication fabric' encompasses a vast variety of logic whose generic responsibility is to transfer data from one point to another within a system. Enroute, it may change some data field(s) for the purpose of arbitration, scheduling, routing or other book-keeping, but involves no heavy computation. Examples of such logic range from local circuitry like a hardware scoreboard to system-wide circuit like a network-on-chip.

Design and analysis of communication fabrics have matured over decades of practice, but very little attention has been given to its formal modeling. Traditionally, there exist various formalisms like Petri nets [Murata, 1989] and data-flow networks [Arvind and Culler, 1986] for modeling communication-centric hardware systems, but they have witnessed only a modest adoption in the main-stream hardware industry. Recently, scientists from Intel's Strategic CAD Laboratory proposed a new formal model called *Executable Micro-Architecture Specification*, xMAS in short, which is particularly suitable for modeling communication fabrics and has promises for seamless adoption in the hardware engineering community [Chatterjee *et al.*, 2010], [Chatterjee and Kishinevsky, 2010a], [Gotmanov *et al.*, 2011]. It has a simple but rigorous formal semantics which makes it amenable to formal analysis and yet it is very close to the way the hardware engineers are trained to think. This is our choice of formal model for expressing and analyzing communication fabrics for their liveness properties. In this chapter, we present xMAS in detail. Readers already familiar with xMAS may skip this chapter, but a good understanding of the examples presented in Section 3.2 is necessary to grasp the subsequent chapters.

The chapter is organized as follows: we begin with a detailed discussion on the structural components of xMAS and their formal semantics (Section 3.1). We then illustrate a variety of example fabrics modeled in xMAS (Section 3.2). These examples are used as benchmarks in all our subsequent experiments. We conclude this chapter with Section 3.3 where we draw a critical comparison between xMAS and other prevailing formalisms like Petri nets and data-flow networks.

3.1 Executable Micro-Architecture Specification (xMAS)

The core observation that Intel's researchers made while developing xMAS was that communication fabrics are essentially 'networks of finite FIFO buffers, with intervening glue logic'. In most of the cases, this glue logic can be described with a small number of characteristic primitives. xMAS is, therefore, simply described as a library of a small number of structural components [1] that includes **finite FIFO buffer**, **source** and **sink** of data items, synchronization primitives **fork** and **join**, **function**, **arbiter** and **switch**.

Each component of xMAS is defined as a synchronous Boolean circuit. FIFO buffer, source, sink and arbiter have sequential components while the rest are defined as combinational circuits. A communication fabric is constructed by stitching instances of different components together. A fabric thus constructed automatically becomes a synchronous, sequential Boolean circuit triggered by a single global clock. The timing model follows the synchronous model of time as in [Benveniste *et al.*, 2003]. For the ease of high-level modeling and informal exchange of designs, each component is represented with a visual symbol as shown in Figure 3.1. Logical definition of individual components are presented below. For further detail, see [Chatterjee *et al.*, 2010].

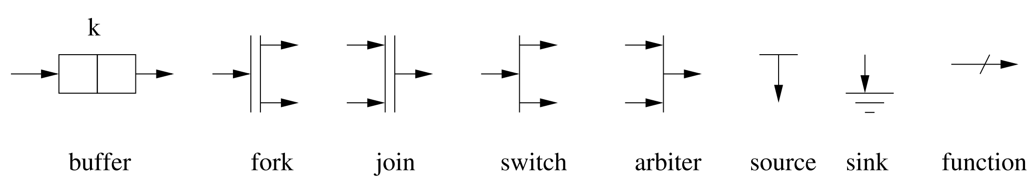


Figure 3.1: xMAS symbols for structural components of communication fabrics

3.1.1 FIFO buffer

Consider a synchronous FIFO queue with a standard interface comprising a read port and a write port as shown in Figure 3.2. The queue has two parameters: size k (the number of elements it can contain) and a type τ (the type of elements it can contain). To compose two instances of such a queue back-to-back without any extra glue logic, xMAS proposes slightly modified interface signals for each FIFO buffer as described below. Figure 3.3 shows composition of two FIFO buffers with such a modified interface definition.

o.data := read_data write_data := i.data
o.irdy := **not** is_empty write_en := i.irdy

[1] we use terms *components*, *structural components* and *primitives* interchangeably in this dissertation

i.trdy := **not** is_full read_end := o.trdy

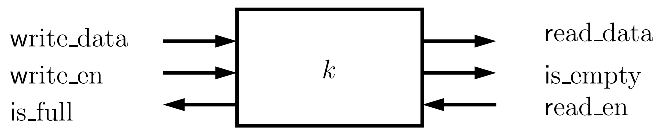


Figure 3.2: A synchronous FIFO queue with a write port (on the left) and a read port (on the right). The queue can store k data elements. In each clock cycle, if the queue is not full, a new element may be inserted; and if the queue is not empty, the oldest element may be removed. read_data exposes the oldest element if the queue is not empty. If the queue is empty, the incoming data appears at the output one cycle later.

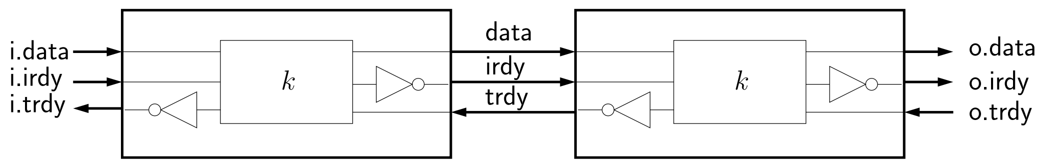


Figure 3.3: Back-to-back composition of two buffers with modified interface signals

3.1.1.1 Channel

Note that input and output ports of FIFO buffers in Figure 3.3 comprises of three signals: *data* (a 'bit vector' signal), *irdy* (a single bit signal, for initiator ready) and *trdy* (another single bit signal, for target ready). This triplet of signals is called a *channel*. Channels are the only communication mechanism in the xMAS framework. A channel always connects two ports: an *initiator*, and a *target*. The data and irdy signals go from initiator to target and the trdy signal go from target to initiator. irdy indicates that the initiator is ready to send data and trdy indicates that the target is ready to accept data. Data are transferred exactly on those clock cycles when both irdy and trdy are true. Each channel has a type τ which indicates the type of data it carries. Channels induce types on the ports they connect to. For example, both i and o ports of a queue have type τ and this is denoted by $i, o : \tau$. In subsequent diagrams, we use only a single line to designate a channel which essentially encapsulates all the three signals associated with it. For example, Figure 3.4 is a simplified version of Figure 3.3 which captures all functional information about the system in a succinct way.

The labels k on the buffers in Figure 3.4 denote that the buffers are of size k. In the sequel, we will often drop this label from the diagrams when no ambiguity is ensued or when the buffer sizing information is not important for the context.

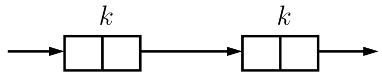


Figure 3.4: A succinct representation of two buffers connected back-to-back from Figure 3.3

3.1.2 Sources and Sinks

A *source* is a primitive which is parameterized by a constant expression $e : \alpha$. At each cycle, it non-deterministically attempts to send a packet e through its output port. A source has a single output port $o : \alpha$ and is governed by the following equations:

o.irdy := oracle **or pre**(o.irdy **and not** o.trdy)
o.data := e

where **pre** is the standard synchronous operator that returns the value of its argument in the previous cycle and the value 0 in the first cycle; and oracle is an unconstrained primary input that is used to model the non-determinism of the source in the synchronous model. o.irdy is persistent, i.e. once a source makes a value available on the channel, it preserves that value until a transfer.

A *sink* is a component which non-deterministically consumes a packet. It has one input port i and is characterized by the following equation:

i.trdy := oracle **or pre**(i.trdy **and not** i.irdy)

3.1.3 Synchronization

A *fork* is a primitive with one input port $i : \alpha$ and two output ports $a : \beta$ and $b : \gamma$ parameterized by two functions $f : \alpha \rightarrow \beta$, and $g : \alpha \rightarrow \gamma$. Intuitively, a fork takes an input packet and creates a packet at each output. It coordinates the input and outputs so that a transfer only takes place when the input is ready to send and both the outputs are ready to receive. Formally,

a.irdy := i.irdy **and** b.trdy	a.data := f(i.data)
b.irdy := i.irdy **and** a.trdy	b.data := g(i.data)
i.trdy := a.trdy **and** b.trdy	

A *join* is the dual of a fork. It has two input ports $a : \alpha$ and $b : \beta$ and one output port $o : \gamma$. It is parameterized by a single function $h : \alpha \times \beta \rightarrow \gamma$. Intuitively, a join takes two input packets (one at each input) and produces a single output packet. It coordinates the input and outputs so that a transfer only takes place when the inputs are ready to send and the output is ready to receive. Formally,

a.trdy := o.trdy **and** b.irdy	
b.trdy := o.irdy **and** a.irdy	
o.irdy := a.irdy **and** b.irdy	o.data := h(a.data, b.data)

3.1.4 Function

A *function* is an xMAS primitive that transforms data. It is parameterized by two types α and β and a function $f : \alpha \rightarrow \beta$. It has an input port $i : \alpha$ and an output port $o : \beta$. It is characterize by the following combinational equations:

o.irdy := i.irdy   o.data := f(i.data)
i.trdy := o.trdy

3.1.5 Switching

A *switch* is a primitive to route packets in the network. It consists of one input port i and two output ports a and b, all of type α. It is parameterized by a switching function $s : \alpha \rightarrow Bool$. Informally, the switch applies s to a packet x at its input, and if $s(x)$ is true, it routes the packet to port a and otherwise it routes it to port b. Formally,

a.irdy := i.irdy **and** s(i.data)   a.data := i.data
b.irdy := i.irdy **and not** s(t.data)   b.data := i.data
i.trdy := (a.irdy **and** a.trdy) **or** (b.irdy **and** b.trdy)

3.1.6 Arbitration

Arbitration is modeled by a *merge* primitive that selects one packet among multiple input ports and one output port. Requests for a shared resource are modeled by sending packets to a merge and a grant is modeled by the selected packet. A complete definition of a two-input merge that has two input ports $a : \alpha$ and $b : \alpha$ and one output $o : \alpha$

o.irdy := a.irdy **or** b.irdy
o.data := a.data
a.trdy := u **and** o.trdy **and** a.irdy
b.trdy := **not** u **and** o.trdy **and** b.irdy

$$u := \begin{cases} 1 & \textbf{if } \text{a.irdy } \textbf{and not } \text{b.irdy} \\ 0 & \textbf{if not } \text{a.irdy } \textbf{and } \text{b.irdy} \\ \textbf{not pre}(u) & \textbf{if pre}(\text{o.irdy } \textbf{and } \text{o.trdy}) \\ \textbf{pre}(u) & \textbf{otherwise} \end{cases}$$

3.1.7 Persistence Property of xMAS Components

xMAS components are so defined that the associated channels hold a property called *persistence*. Informally, a signal is persistent means that once the signal is asserted by the initiator agent, it remains asserted until it is properly served by the target agent. For any xMAS channel u, both its irdy and trdy signals are persistent. Persistence of these two signals of channel u is termed as *forward persistence* and *backward persistence* and denoted with FwdPersistence(u) and BwdPersistence(u) respectively. Formally, these two properties of channel u can be defined in LTL as follows:

$$\text{FwdPersistence}(u) \triangleq \ \mathbf{G}((u.irdy.\neg u.trdy) \Rightarrow \ \mathbf{X}u.irdy)$$

$$\text{BwdPersistence}(u) \triangleq \ \mathbf{G}((u.trdy.\neg u.irdy) \Rightarrow \ \mathbf{X}u.trdy)$$

Persistence is an important property that every xMAS channel satisfies. It greatly simplifies behavior of xMAS fabrics, significantly limits potential deadlocks and allows for simpler definition of channel liveness. It helps in compositional analysis of fabric behaviors and their liveness analysis with intermediate safety properties.

3.2 Benchmarks

We illustrate how xMAS primitives are used to construct communication fabrics with the help of few examples. These examples are taken from industrially relevant applications and are used as benchmarks in our subsequent experiments. We discuss the following examples in order: *credit logic*, *virtual channel*, *virtual channel with buffer*, *virtual channel with ordering*, *master/slave communication* and *hardware scoreboard*.

3.2.1 Credit Logic

Credit-based flow-control is a popular technique of switch-level flow control, especially in wormhole routing systems. Figure 3.5 illustrates an xMAS implementation of the basic idea of credit-based flow-control mechanism where a source (S_1) wants to send flits [2] to a sink (S_2) through a buffer (B_2), and the flow of flits across B_2 is controlled by the $credit\_logic$ sub-circuit. Main purpose of $credit\_logic$ is to allow only a restricted number of flits to enter the system which it is guarding (in this case, the buffer B_2).

3.2.2 Virtual Channel

Consider the model shown in Figure 3.6. It is an implementation of *virtual channel* in xMAS. Here a single physical channel (channel e) is shared between two sources (sources A_1 and A_2). Virtual channel is a fundamental building block of on-chip communication networks which was invented to mitigate the head-of-line (HOL) blocking problem in wormhole switching. See texts like [Dally and Towles, 2003], [Duato *et al.*, 2002] for details. Figure 3.6 shows how two

[2] A flit (flow-control unit) is a unit of data transfer in communication fabrics, see [Dally and Towles, 2003] for details.

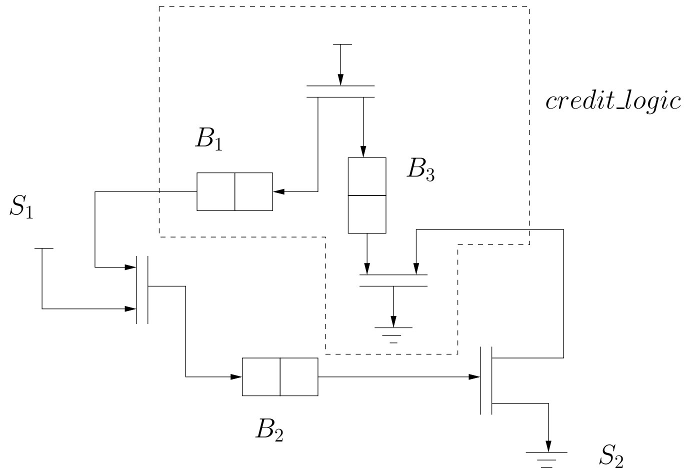


Figure 3.5: Credit mechanism

sources A_1 and A_2 share the virtual channel e to transfer flits to their respective sinks ($sink_1$ and $sink_3$ respectively) while credit logic blocks CL1 and CL2 control flow of flits from source A_1 and A_2 respectively. When source A_1 wants to send a flit to $sink_1$, it makes a request on *channel* a_1. If buffer B_1 has a token, this request is forwarded to channel d_1. Based on various conditions like whether channel d_2 is also making a concurrent request or whether buffer B_3 has an empty slot, the arbiter decides when to issue a grant against the request on channel d_2. This grant is transmitted to source A_1 which allows A_1 to complete the flit transmission. Similar steps take place when A_2 wants to send a flit to $sink_3$ and makes a request on channel a_2. We refer to the model of Figure 3.6 as $\mathcal{VC}$ in the subsequent chapters.

3.2.3 Virtual Channel with Buffer

In different applications, virtual channel needs to store flits temporarily before it forwards them to their respective destinations. This is implemented by putting a temporary buffer on the shared channel itself. An xMAS implementation is shown in Figure 3.7. We refer to this model as $\mathcal{VCB}$. Structurally, it is same as $\mathcal{VC}$ with the only difference of the presence of buffer B_{ch} which splits channel e into two channels e_1 and e_2.

3.2.4 Virtual Channel with Order

Suppose we have the following ordering restriction on flits travelling in a virtual channel:

> *an A_1-flit can be sunk only if all A_2-flits that came before it on channel e have been sunk.*

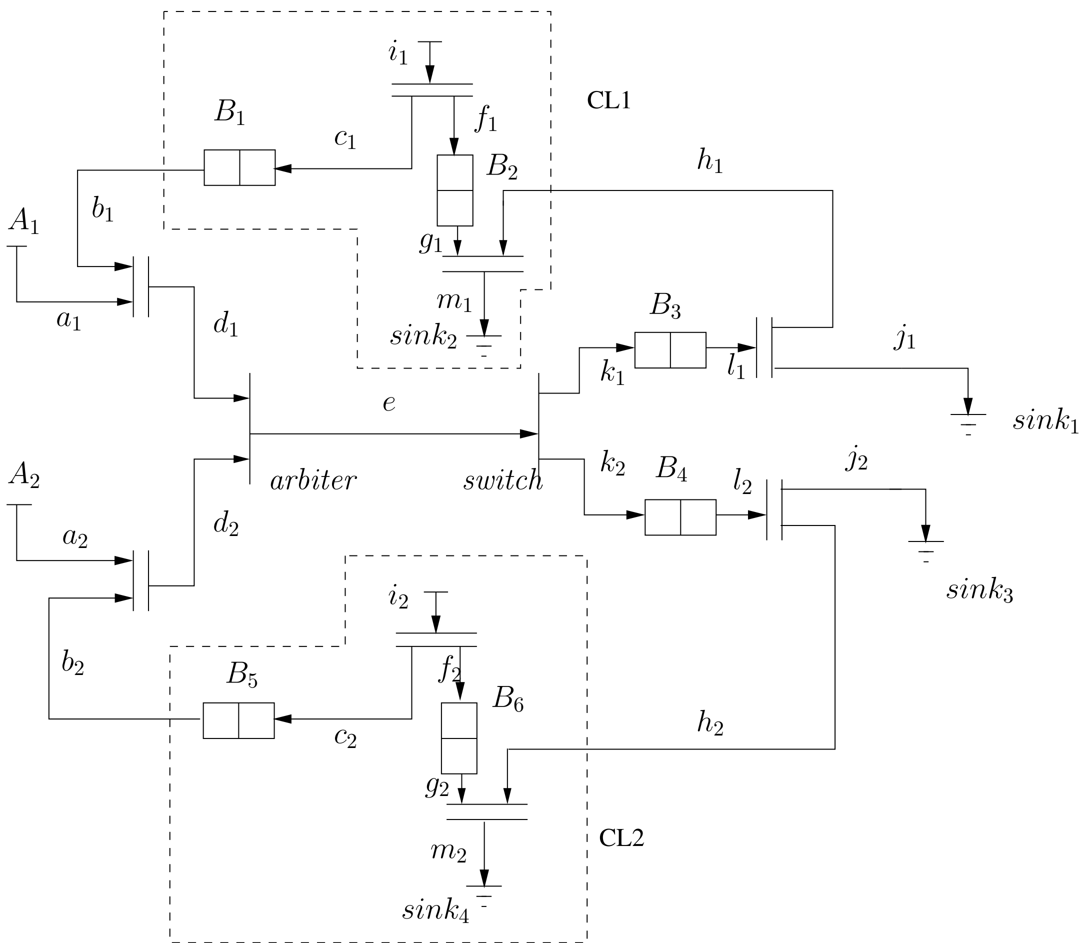


Figure 3.6: Virtual channel ($\mathcal{VC}$)

e is called an *ordering point*. Such one-way ordering restrictions are often needed to implement cache coherence protocols, or to guarantee producer-consumer ordering for software yet avoiding deadlock as would have been caused by total ordering. Figure 3.8 shows how this may be implemented using xMAS primitives. We call this model $\mathcal{VCO}$. In this implementation, as shown in Figure 3.8, all buffers except B_5 can hold two flits. B_5 can hold four flits instead. Another speciality of B_5 is that it can hold both $type(A_1)$ and $type(A_2)$-flits, whereas each of the other buffers holds either $type(A_1)$, or $type(A_2)$ flits, but not both. For further detail, see [Chatterjee *et al.*, 2010].

3.2.5 Master/Slave Communication

Figure 3.9 shows two agents *P* and *Q* communicating over a trivial fabric composed of six queues. Two types of flits are circulating in this fabric, viz. *req* (request type), and *rsp*

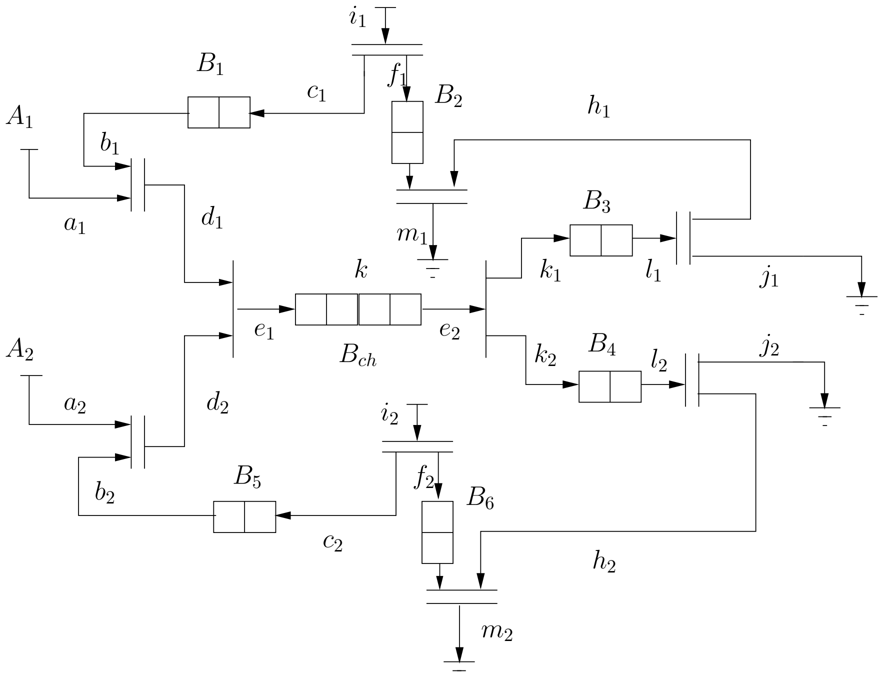


Figure 3.7: Buffered virtual channel ($\mathcal{VCB}$)

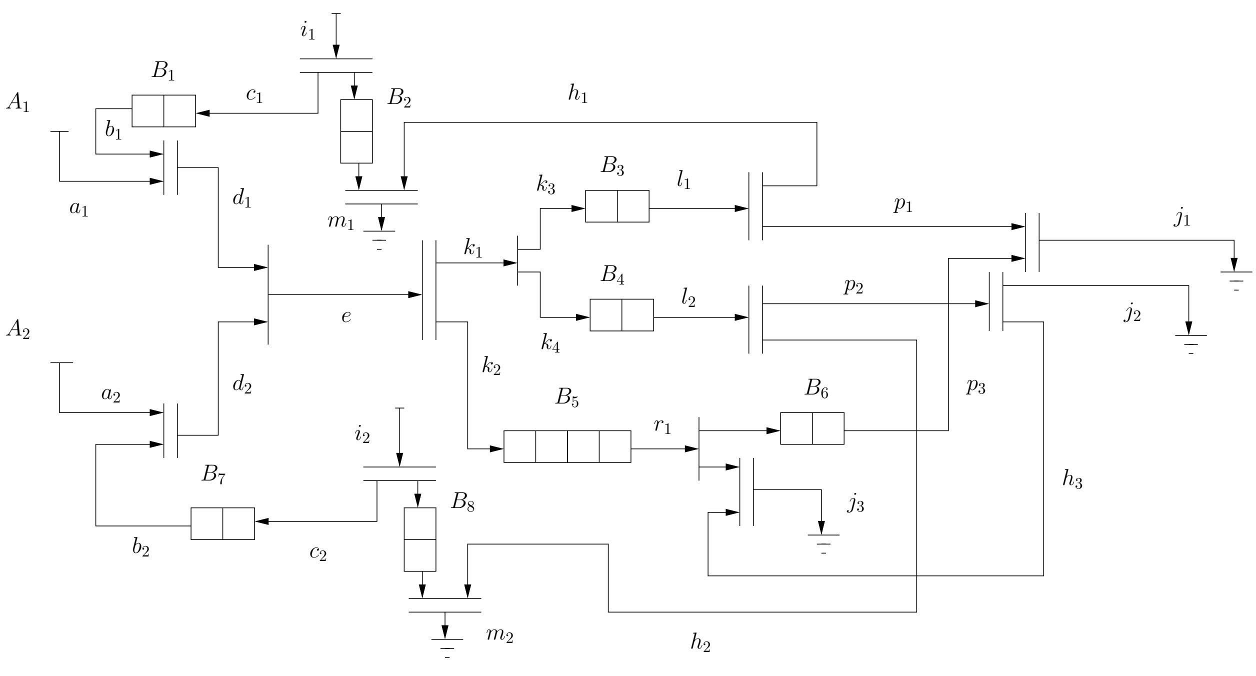


Figure 3.8: Virtual channel with ordering ($\mathcal{VCO}$)

(response type). Each agent creates new requests for the other agent. When an agent receives a request, it produces a response (by changing the packet type using a function $x \mapsto rsp$) after a non-deterministic delay. The response is sent back to the original agent where it is sunk when the sink is ready to receive it. Thus each agent behaves like a master that produces requests, and responses, and a target that consumes responses, and requests. Communication between agents is done through the virtual channels. Consider agent P as example. It sends requests, and responses to agent Q through the shared channel, and the data transfer queue B_4, and then to two ingress queues B_7, and B_8, one per message type. An arbiter modeled by the merge primitive selects fairly between *req* and *rsp* messages that are exposed to arbitration only if they have credit tokens inside the corresponding credit queues B_1, and B_2. Credits are initialized inside the credit counters B_6, and B_9 to the values equal to the sizes of the ingress queues B_7, and B_8, i.e. to k. Credits are returned through fabric credit queues B_3, and B_5. We call this model $\mathcal{MS}$.

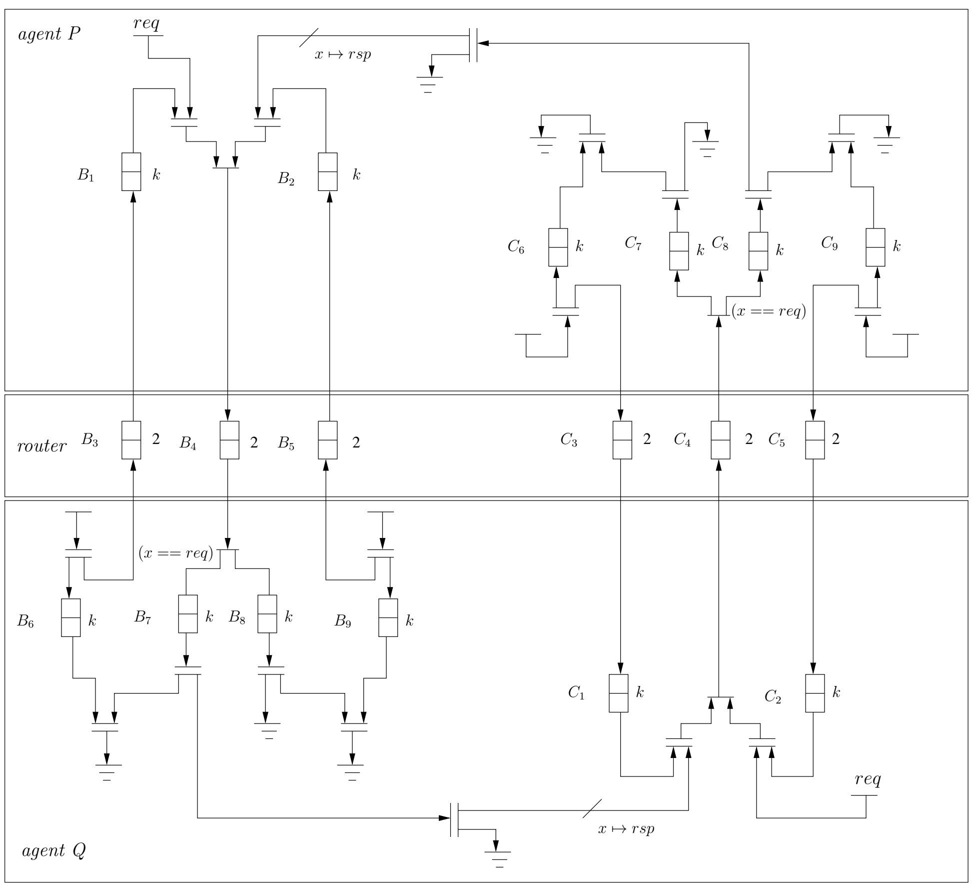


Figure 3.9: A pair of agents communicating over a simple fabric ($\mathcal{MS}$)

3.2.6 Hardware Scoreboard

Figure 3.10 shows how a two-entry scoreboard may be modeled using the xMAS primitives. An incoming transaction on the left needs to obtain a tag before it can enter the scoreboard. Different tags are used to distinguish different in-flight transactions in the scoreboard. In this example, the scoreboard supports two simultaneous in-flight transactions, and hence there are two tag sources. These tag sources are modeled using credit logic. Once the transaction enters the scoreboard it competes with the other transaction (if there is one) to enter the first phase of processing. The results of this phase may return out of order: tags are used to match a result with the corresponding transaction in the scoreboard. Once the result of the first phase is returned, the transaction moves on to the second phase. After the second phase is done, the transaction becomes eligible for retirement. When it wins arbitration, it retires and releases its tag which is then recycled for use by a future transaction. For details, see [Chatterjee *et al.*, 2010]. We call this model $\mathcal{SB}$.

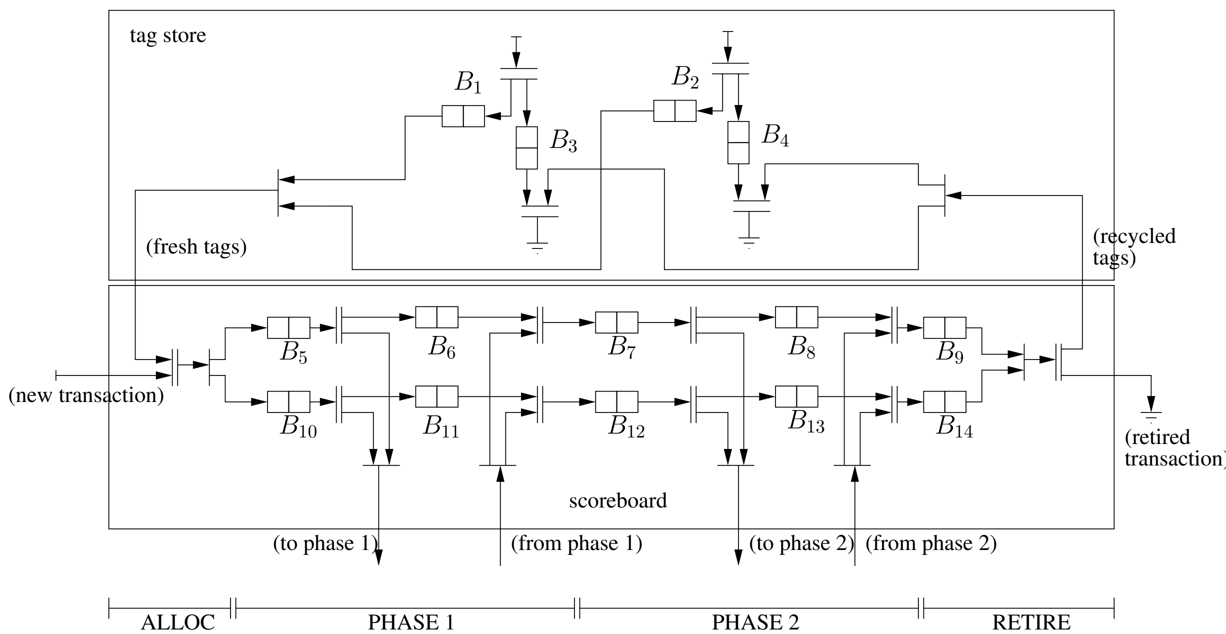


Figure 3.10: A two entry scoreboard ($\mathcal{SB}$)

3.3 xMAS in Perspective

3.3.1 From xMAS to bit-level netlist

As we mentioned before, an xMAS fabric is constructed by stitching various instances of the primitives together. This connection scheme can be represented textually, as well as diagrammatically. Thanks to the precise logical semantics of the primitives, such a connection scheme can easily be translated into Verilog or VHDL or any other hardware description language (HDL) and from there, a bit-level netlist can easily be generated using any existing compiler. This quick-and-easy translation of a micro-architectural description into a bit-level netlist enables us to use state-of-the-art SAT solvers and bit-level verification tools for verifying architecture level properties. We have developed our own Verilog implementation of xMAS library closely following the semantics defined in [Chatterjee *et al.*, 2010]. We have also developed a compiler that can read plain textual descriptions of connection schemes and generate Verilog descriptions. In our framework, we convert these Verilog descriptions into bit-level netlists by invoking our in-house tool veriABC. It uses Verific, a commercial Verilog front-end analyzer, for translation and elaboration into netlists. Finally, we use our bit-level verification tool ABC to perform verification experiments on them. This seamless integration of micro-architecture level modeling with the main-stream hardware verification flow is the major strength of xMAS formalism and this makes xMAS more attractive than some of the prevailing formalisms as discussed below.

3.3.2 Comparison with Petri nets

Petri net is an extensively studied model of computation, suitable for modeling and analysis of concurrent, and distributed systems, both in software and hardware. A huge volume of literature is available on this topic. [Murata, 1989] is an excellent introduction to Petri nets and their applications. Over the past few decades, different variations of the basic Petri net have been proposed and used in modeling various complex scenarios in hardware design. However, in spite of extensive study in academia, this formalism is hardly used in practice for developing large, complex systems. We may identify the following reasons for poor adoption of Petri nets in the industry:

- first, most hardware engineers feel comfortable thinking about a design in terms of the operational semantics of their choice of HDL - Verilog or VHDL most of the time. Expressing a design in a different formal model like Petri net and then bit-blasting it often seem counter-productive to them.

- second, legacy designs and codes play a significant role in this mind-set. Design houses have huge repertoire of designs already coded in HDL's like Verilog or VHDL. It is practically impossible to re-model them in some other formalism that share little syntactic similarity with the HDL's.

- third, the most successful analysis tools for Petri nets are developed mainly for the simplest class of nets. For effective modeling of real-life hardware systems, we need more expressive families of Petri nets, like *colored Petri net*. Unfortunately, mathematical properties of these advanced families are rather weak, less studied and/or not so well-known. There is not much of tool support and human expertise available for analysis of systems modeled in such advanced formalisms.

On the contrary, xMAS is a trivial syntactic extension over the basic HDLs that a hardware engineer is acquainted with. Adapting xMAS requires no extra learning, analysis or tool development.

3.3.3 Comparison with Data-flow networks

Data-flow networks are widely used models of computation for design of embedded systems. However, they are not so popular in the main-stream hardware industry. While the design automation tools for data-flow networks meet the needs of embedded system designers, they hardly provide any extra benefit to the general-purpose hardware engineers. The most 'practically effective' data-flow models, viz. *synchronous data-flow* (SDF) [Lee and Messerschmitt, 1987] or *cyclo-static data-flow* (CSDF) [Bilsen *et al.*, 1995], do not capture all the modeling needs for general-purpose hardware. *Boolean data-flow* (BDF) [Buck, 1993], which is capable of modeling Boolean conditions, switching, merging etc., is perhaps the most expressive variation of data-flow models that suites the purpose of general-purpose hardware design.

xMAS models do have syntactic similarities with BDF models, but there are several semantic differences that make BDF more expressive than xMAS. Some differences are outlined below:

- xMAS has finite memory, whereas in the standard dataflow models, queues are a-priori unbounded. Some models like BDF can simulate Turing machines using this unbounded memory capability. On the other hand, finite FIFOs can easily be simulated in dataflow models by adding *backward queues* to control writing to a *forward queue* (so the number of initial tokens in the backward queue represents and controls the size of the forward queue).

- xMAS is a synchronous model, whereas dataflow models are asynchronous: each process proceeds at its own pace, and only needs to wait/synchronize with other processes when it needs to wait for data on input queues or when output queues are full (which is simulated by waiting for data on the backward queue). These synchronization features are also part of xMAS, but still the latter is a fundamentally synchronous model.

- xMAS has non-deterministic components (eg. sources, sinks) whereas in Kahn Process Networks and their subclasses like BDF, SDF, etc, processes are deterministic.

In spite of being more expressive than xMAS, however, BDF does not possess any particularly useful mathematical property that the other variations of data-flow models possess. For hardware communication fabric design, BDF's extra expressive power does not offer any extra benefit over xMAS. Analysis of hardware modeled as BDF rely on the same Boolean techniques (like model checking) as xMAS models do. The latter, on the other hand, provides neat modeling primitives for finite FIFO networks compared to the cumbersome 'backward queue techniques' of BDF models. This offers xMAS a competitive advantage over BDF in practice. It might be interesting to perform a rigorous comparative analysis between Boolean data-flow and xMAS as models of computation, but we leave this discourse outside the scope of this dissertation.

Chapter 4

Bug Hunting for Liveness

4.1 Introduction

Counterexample generation, an important step in formal verification, produces a trace demonstrating why the design does not meet a specification. Automatic generation of counterexamples is one of the most useful features of model checkers. It is most useful during the early design cycles when the design has not matured and contains many bugs. A model checker helps focus on bugs in the designs through the counterexamples it generates. The two most successful technologies for counterexample generation for safety properties are random simulation and bounded model checking. In this chapter, we describe how we use them to generate counterexamples for a class of liveness properties called *stabilization properties*. For this, we use a construction called *liveness-to-safety transformation*, originally proposed by Biere *et al.* [Schuppan and Biere, 2004]. The chapter is organized as follows: we begin with a preliminary discussion in Section 4.2. Then we present the liveness-to-safety conversion scheme in Section 4.3 and conclude the chapter with experimental results presented in Section 4.4.

4.2 Preliminaries

The most intuitive notion of stabilization states that the system will always reach a particular state, and will stay there forever, no matter which state the system started from, or which path it took. More generally, stabilization means that the system will eventually reach and stay within a given subset of states. Also, stabilization may denote conditions on the input and output signals of a system when it attains a stable state. For recent applications of stabilization properties, see [Cook *et al.*, 2011] and [Gotmanov *et al.*, 2011]. Below we review some basic linear temporal logic (LTL) terminology and formally define stabilization using LTL.

4.2.1 LTL, Model Checking and Stabilization Property

In this chapter, we assume reader's familiarity with LTL syntax and semantics, basic model checking algorithms and related terminology like Kripke structures, Büchi automata etc. For further details, see [Clarke *et al.*, 1999]. In our current context, we use LTL properties $\mathbf{GF}p$ and $\mathbf{FG}p$ and overview their semantics here: let π be a path in some Kripke structure K; $\pi \models_K \mathbf{G}p$ means property p will hold on every state along π; $\pi \models_K \mathbf{F}p$ means property p will hold eventually on some state along π; $\pi \models_K \mathbf{GF}p$ means p will hold along π infinitely often and $\pi \models_K \mathbf{FG}p$ means p will hold eventually on π forever. Temporal operators $\mathbf{F}$ and $\mathbf{G}$ are dual (i.e. $\mathbf{F}p \equiv \neg\mathbf{G}\neg p$), operators $\mathbf{FG}$ and $\mathbf{GF}$ are also dual (i.e. $\mathbf{FG}p \equiv \neg\mathbf{GF}\neg p$).

Definition 3 (GF-atom)**.** An LTL formula of the form $\mathbf{GF}p$ or $\mathbf{FG}p$, where p is some atomic proposition or some Boolean formula involving atomic propositions only, will be called a 'GF-atom'.

Stabilization properties are defined as the family of LTL formulas that are Boolean combinations of GF-atoms. Formally:

Definition 4 (Stabilization Property)**.** The set of stabilization properties is the syntactic subset of LTL defined as:

- any GF-atom is a stabilization property
- if ϕ is a stabilization property, then so is $\neg\phi$
- if ϕ and ψ are stabilization properties, then so are $\phi \wedge \psi$ and $\phi \vee \psi$

Example 1. $\mathbf{FG}p$, $\mathbf{GF}p \Rightarrow \mathbf{GF}q$, $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$ and $\mathbf{FG}p \Rightarrow \mathbf{FG}q \vee (\mathbf{FG}r \wedge \mathbf{GF}s)$ are few examples of stabilization properties where p, q, r and s are atomic propositions or Boolean formulas involving atomic propositions only ($a \Rightarrow b$ is the usual shorthand for $\neg a \vee b$). However, $\mathbf{G}(r \Rightarrow \mathbf{F}g)$ is an LTL liveness property but not a stabilization property.

Needless to say, all stabilization properties are liveness properties. However, not all of them specify system stabilization directly. Properties like $\mathbf{FG}p$ and $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$ (or its generalization $\wedge_{i=1}^{k}\mathbf{FG}p_i \Rightarrow \mathbf{FG}q$) are perhaps the most elementary stabilization properties. $\mathbf{FG}p$ means that the system eventually will reach a state from where p will always hold, i.e. the system will eventually 'stabilize' at p. $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$ means that if the system stabilizes at p and also at q (possibly at some other time), then it will eventually stabilize at r. Hence, the semantics of these properties are close to the intuitive notion of stabilization. [Cook *et al.*, 2011] has demonstrated an use and significance of this kind of stabilization properties in the context of biological systems. However, our definition of stabilization captures a broader family of specifications. It includes properties like $\mathbf{FG}p \Rightarrow \mathbf{FG}q \vee (\mathbf{FG}r \wedge \mathbf{GF}s)$ which may look contrived, but for example [Gotmanov *et al.*, 2011] uses many such complicated stabilization properties for compositional deadlock analysis of micro-architectural communication fabrics.

Our definition also includes many properties which are not intended to specify so-called stabilization behavior. For example, $\mathbf{GF}p$ or $\mathbf{GF}p \Rightarrow \mathbf{GF}q$. The main motivation behind considering this broader subset of LTL is that we offer a short-cut liveness-to-safety (L2S) conversion, avoiding Büchi automaton construction, in a uniform way. This uniformity in our treatment comes from the duality between **FG** and **GF** operators. The most significant applications of this class of properties that we have encountered is *stabilization verification* and hence the name is coined for the class. This name is inspired by [Cook *et al.*, 2011].

The class of LTL properties, defined as stabilization properties in this thesis, is a very important class extensively studied by the temporal logic community. It is related to so-called *fairness specifications*. Operators **GF** and **FG** are often called *infinitary operators* [Hojati *et al.*, 1993a] and symbols F^{∞} and G^{∞} are used instead respectively [Emerson, 1990]. The class itself has been called *general fairness constraints* [Emerson and Lei, 1987], [Hojati *et al.*, 1993a]. As shown in [Emerson and Lei, 1987], various notions of fairness like *impartiality* [Lehmann *et al.*, 1981], *weak fairness* [Lamport, 1980](also called *justice* [Lehmann *et al.*, 1981]), *strong fairness* [Lamport, 1980] (also called *compassion* [Lehmann *et al.*, 1981]), *generalized fairness* [Francez and Kozen, 1984], *state fairness* [Pnueli, 1983] (also known as *fair choice from states* [Queille and Sifakis, 1983]), *limited looping fairness* [Abrahamson, 1980] and *fair reachability of predicate* [Queille and Sifakis, 1983] can be expressed by stabilization properties. These properties are used to exclude "unfair" counterexamples in liveness verification in both linear time and (fair) branching time paradigms. For liveness verification, we usually have a liveness property (the actual proof obligation) along with a set of fairness constraints. The liveness property in hand may not be a stabilization property. In that case we may need to construct the product of the system and the Büchi automaton of the (negation of the) liveness property before performing the L2S conversion. Interestingly, for applications like [Cook *et al.*, 2011] and [Gotmanov *et al.*, 2011], the liveness verification obligations fall entirely in the family of stabilization properties. For these applications, the simple L2S scheme proposed in this chapter works. Note that some liveness properties like $\mathbf{G}(request \Rightarrow \mathbf{F}grant)$ are not stabilization properties, but also have a direct L2S conversion [Schuppan and Biere, 2004]. It is, therefore, an interesting question that under what more general conditions does there exist a direct L2S conversion.

4.3 L2S Conversion for Stabilization Properties

It is important to understand that any counterexample to a liveness property (which must be an infinite trace) can be seen as a "lasso" like configuration with a finite handle and a finite loop. A liveness counter-example is a lasso which does not satisfy the property on the loop but satisfies all imposed fairness constraints on the loop.

In general, a liveness problem is converted to a safety problem by adding loop-detection logic and property-detection logic to the product of FSM of the original system and the Büchi automata of the property. The loop-detection logic consists of a set of shadow registers,

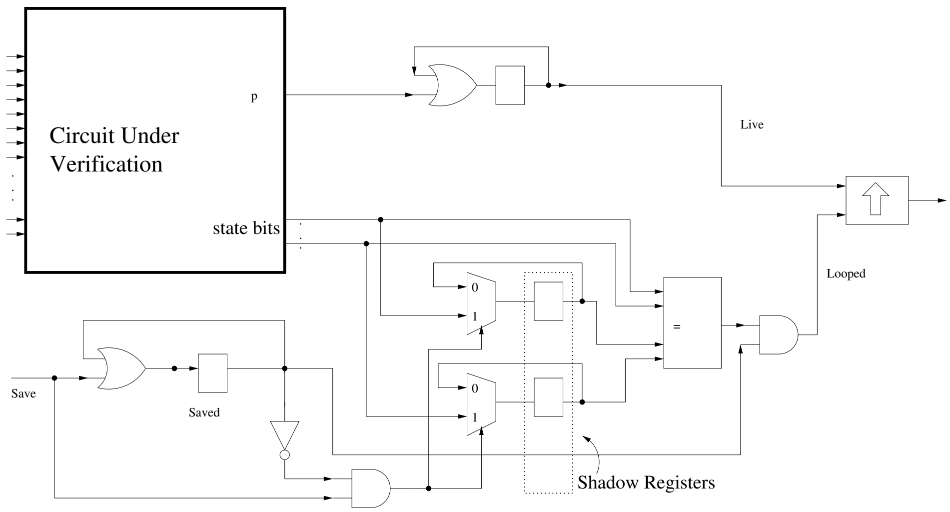


Figure 4.1: Liveness-to-safety transformation for Fp

comparator logic and an 'oracle'. The oracle saves the system state in the shadow registers at a non-deterministically chosen time. In all subsequent time frames, the current state of the system is compared to the state in the shadow registers. Whenever these two states match, the system has completed a loop. The non-deterministic nature of the oracle allows all such loops to be explored. The property verification logic checks if any of the liveness conditions are violated in any such loop and all fairness conditions always hold in the loop. This check is done as a safety obligation. For a more detailed exposition, see [Schuppan and Biere, 2004].

As mentioned before, for some simple properties L2S conversion can be achieved while avoiding explicit Büchi automata construction. This is done by adding more functionality to the property detection logic. As presented in [Schuppan and Biere, 2004], these properties are $\mathbf{F}p, \mathbf{GF}p, \mathbf{FG}p, p\mathbf{U}q, , \mathbf{G}(r \Rightarrow \mathbf{F}q)$ and $\mathbf{F}(p \wedge \mathbf{X}q)$ (see Table 1 of [Schuppan and Biere, 2004]). This approach is reviewed in Figure 4.1 which depicts an L2S converted circuit for verifying the LTL property $\mathbf{F}p$. How this construction works is presented below. In Section 4.3.1 we explain how to extend the same idea of Figure 4.1 for stabilization properties. Instead of presenting the liveness-to-safety conversion through Kripke structure-based representations, we present the idea in terms of circuit construction. The same mechanism handles fairness constraints too which are always stabilization properties. Handling fairness constraints only requires addition of extra logic to the property monitor. For Kripke structure-based descriptions of liveness-to-safety conversion, see [Schuppan and Biere, 2004].

In Figure 4.1, save represents an additional primary input added to the circuit. This plays the role of the 'oracle'. When save is asserted for the first time, the current state of the circuit is saved in the set of shadow registers and register saved is set. saved thus remembers that input save has been asserted and any further activity on save is ignored thereafter. For subsequent time frames, saved enables the equality detector between the current state and the state in the shadow registers. Clearly, signal looped is asserted iff the system has completed a loop. Signal live remembers if signal p has ever been asserted. The safety property that the circuit verifies is, therefore, looped $\Rightarrow$ live. In general, this is looped $\wedge$ fair $\Rightarrow$ live. The block marked with "$\Uparrow$" represents logical implication - the direction of the arrow distinguishes the antecedent signal from the consequent signal.

4.3.1 L2S for Stabilization Properties

In [Schuppan and Biere, 2004], the authors show how to do the L2S conversion for $\mathbf{GF}p$ and $\mathbf{FG}p$, which are GF-atoms. We demonstrate how to extend this to any Boolean combination of GF-atoms using the following example. Formal proof of correctness of this construction is straightforward, thus omitted.

Consider a simple stabilization property ϕ of the form $\mathbf{FG}a \Rightarrow \mathbf{FG}b + \mathbf{FG}c$. An L2S converted circuit for ϕ is shown in Figure 4.2. (For simplicity, we do not show any fairness constraint in the example.) Note that signal live in Figure 4.1 monitors if signal p has ever been asserted. But for verifying $\mathbf{GF}p$, we need to monitor whether signal p has been asserted between the time when saved is set and the time when looped becomes true. Using this fact and the duality between $\mathbf{FG}$ and $\mathbf{GF}$ operators and the Boolean structure $X_a \Rightarrow X_b + X_c$ of ϕ we can derive the circuit of Figure 4.2. Logic that captures the Boolean structure of ϕ is marked with a dotted triangle in Figure 4.2. Therefore for any arbitrary stabilization property, we need to create monitors for individual GF-atoms and a *crown of combinational logic* on top of these monitors that captures the Boolean structure of the property. The following theorem captures the key behavior of such an L2S-converted circuit:

Theorem 1. For any stabilization property ϕ, a safety verification engine will find a counterexample in the L2S-converted circuit constructed using the above method if and only if the original circuit violated ϕ.

(Proof Sketch) Any stabilization property can be transformed into another stabilization property with $\mathbf{GF}$ operators only. Let f be the Boolean structure in the negation of the given stabilization property. The procedure described above will create a monitor that will search for a lasso-loop where f is violated inside the loop. Since the procedure implicitly enumerates all possible cycles in the state space, it will detect a violating cycle if one exists.

4.4 Experimental Results

We implemented our L2S scheme for general stabilization properties in ABC and experimented with several designs of communication fabrics from industry. Our objective was to verify all stabilization properties defined for every structural primitive of the xMAS framework [Gotmanov *et al.*, 2011]. The properties, though local to each component, are verified in

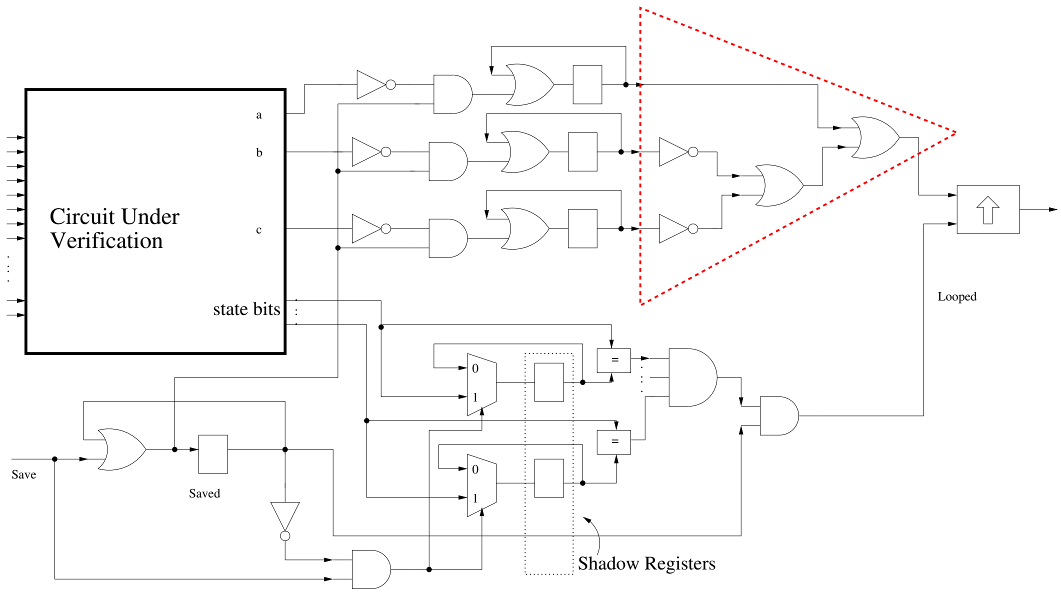


Figure 4.2: L2S for stabilization property $FGa \Rightarrow FGb + FGc$

the context of the whole design in order to avoid explicit environmental modeling. BLIF models of the communication fabrics were generated by the xMAS compiler [Chatterjee *et al.*, 2010] from high-level C++ models. The L2S monitor logic was then created by ABC on these BLIF models. The xMAS compiler also generates SMV models from C++ models so that the LTL encoding of the stabilization properties can be verified directly on the SMV models using the NuSMV model checker.

We found that the ABC based L2S implementation has much better scalability than NuSMV. NuSMV can solve only toy designs while on the large designs of interest, it fails to produce a result. On the other hand, our tool works well even on large designs. For most cases, it produces a result almost immediately. For a few cases, initial trials could not produce a proof, but with the latest version of ABC using simplification, abstraction, speculative reduction and property directed reachability (PDR) analysis [Bradley, 2011], the proofs were completed. This observation supports the premise that the use of highly developed safety techniques can pay off for liveness verification.

Experimental results are shown below. Among all the local properties that the xMAS compiler generated, we provide results for the most challenging one. Call this property ψ; it is defined for a FIFO buffer and has the following LTL form

$$\psi := \mathbf{FG}(\neg a) \Rightarrow \mathbf{FG}(\neg b) \vee \mathbf{FG}(c)$$

where a, b, and c are appropriate design signals (i.e. interface signals of a FIFO buffer). Table 1, 2 and 3 compare the performance of ABC with NuSMV on small examples. These examples are instances of communication fabrics or sub-modules thereof and are explained

Prop #	ABC	NuSMV
	(sec)	(sec)
0	0.25	0.115
1	0.05	0.14
2	0.02	0.09

Table 4.1: $\mathcal{CL}$

Prop #	ABC	NuSMV
	(sec)	(sec)
0	0.09	33.23
1	0.07	31.8
2	0.06	39.57
3	0.03	16.46
4	0.5	41.37
5	0.03	16.89

Table 4.2: $\mathcal{VC}$

Prop #	ABC	NuSMV
	(sec)	(sec)
0	0.03	431.5
1	0.12	379.59
2	0.8	471.36
3	0.8	385.67

Table 4.3: $\mathcal{MS}$

in full detail in [Chatterjee and Kishinevsky, 2010b]. $\mathcal{CL}$ and $\mathcal{VC}$ (Table 1 and 2 respectively) are designs corresponding to Figure 4 and 5 of [Chatterjee and Kishinevsky, 2010b] and $\mathcal{MS}$ (Table 3) is a much simpler version of the design shown in Figure 6 of [Chatterjee and Kishinevsky, 2010b]. Note from the tables how the performance of NuSMV degrades even for small designs. For large designs, NuSMV could not finish for any single instance of ψ.

Since ψ is defined for a FIFO buffer and the xMAS compiler created one instance of ψ for each FIFO buffer, the number of ψ instances is the same as the number of FIFO buffers. For example, the designs corresponding to Table 1, 2 and 3 above have 3, 6 and 4 FIFO buffers respectively.

We also experimented on two large communication fabrics of practical interest [Chatterjee and Kishinevsky, 2010b], [Gotmanov *et al.*, 2011]. One has 20 buffers and the other has 24 buffers. 19 out of 20 of the first design and 23 out of 24 from the second design were proved by ABC by a light-weight interpolation engine within a worst case time of 5.83 seconds (most were proved in less than a second). Light-weight interpolation could not prove one instance from each design. These were proved using advanced techniques from ABC's arsenal of safety verification algorithms. For example, ABC took a total of 217.2 seconds to prove one of these harder properties. In this time span, ABC first did some preliminary simplification, then it tried interpolation, BMC, simulation and PDR in parallel for a time budget of 20 seconds. But this attempt failed and it moved on to further simplification by reducing the design using localization abstraction and speculation. It ran interpolation, BMC, simulation, BDD-based reachability and PDR engines in parallel both after abstraction and speculation, using an elevated time budget of 100 seconds and 49 seconds respectively. The iteration after abstraction could not prove the property, but the iteration after speculation managed to prove it with the PDR engine, which produced the final proof in 7 seconds.

Chapter 5

Efficient Proof Of Liveness

5.1 Introduction

Liveness properties offer a concise way of specifying reactive behaviors like something 'good' will eventually happen. They offer a way of writing end-to-end specifications without worrying too much about the internal details of a design. Unfortunately, traditional model checking algorithms for such specifications involve computationally expensive formulations and suffer from scalability problems. Typically, these algorithms attempt to enumerate all strongly connected components (SCC) of the state transition graph of the design. Thereby, they try to rule out the existence of a counterexample to the liveness property, but end up being unscalable for large designs.

However, it has been observed that real-life designs often carry *intermediate hints* that may be used to construct deductive proofs of liveness without invoking SCC analysis. We collectively call such alternative hint-based approaches as *ranking-oriented approaches* for proving liveness. These proofs are usually much more scalable compared to their SCC-oriented counterparts. They also reveal more information about the state-space, which may act as *certificates of liveness* for a correct system. SCC-oriented algorithms can return counterexamples, but these usually are not human-readable certificates of correctness. On the down side, it is quite challenging to devise a completely automated proof engine that will discover necessary hints and eventually construct a deductive proof of liveness for an arbitrary system. In this chapter, we address this challenge in the context of bit-level liveness verification of communication fabrics.

We present four alternative ways of addressing the challenge. The first three 'SCC-analysis avoiding' methods are heuristic in nature and tailored for the particular application domain. They use *moderate insights* from designers as intermediate hints and efficiently produce proofs of liveness for benchmarks that are challenging for SCC-oriented algorithms. These three methods involve the following three techniques, respectively:

1. Breaking down a liveness property into *intermediate safety properties* (Section 5.4)

2. Discovering a *well-founded characterization* of the underlying state-space (Section 5.5)

3. Use of *skeleton independent proof heuristics* (Section 5.6)

A common thread of these three approaches is the use of *safety verification technology for liveness verification*. In the previous chapter, we demonstrated how a liveness problem can be translated into an equi-satisfiable safety problem on which safety tools can be used to prove liveness. While this is a viable method, L2S translation incurs a blow up of the state-space and stresses the underlying safety verification engines. This makes L2S practically ineffective for *proving* liveness properties for large designs, though it remains effective for bug-hunting through BMC or simulation. To mitigate this problem, we explore how to exploit specific characterizations of the state-spaces. The three techniques mentioned deal with different ways of expressing such characterizations as safety properties and then use a safety engine to prove them.

While the safety properties, derived in the above three methods, can be proved efficiently, the overall methods are heuristic in nature, work for a particular set of problems and require manual intervention. We end this chapter by presenting a fourth method based on k-LIVENESS (Chapter 5.7). k-LIVENESS is a new algorithm proposed by Koen Claessen and Niklas Sörensson in [Claessen and Sörensson, 2012]. It has outperformed all the existing state-of-the-art liveness verification algorithms in the latest model checking competition. It is a general proof technique that works for any ω-regular property on any sequential circuit. We extend this technique by introducing some dedicated constructions, which enhance its performance on our benchmarks, yet these are general and may be useful for other situations as well. The k-LIVENESS approach also relies on the use of off-the-shelf safety verification engine for liveness verification, but in a clever way.

As benchmarks, we have chosen a family of designs that work on *credit-based flow-control* principle (see Section 5.2 for details). On the specification side, we focus on a particular liveness property called *response* (see Section 5.3 for details). We begin with detailed discussions on both the credit mechanism and the specification in the following two sections, then present the four methods mentioned above in the subsequent sections.

5.2 Credit Mechanism and Buffer Relations

'Credit-based flow-control' is a switch-level flow-control technique, widely used in various communication fabrics for avoiding deadlock, especially in wormhole switching systems (see Chapter 13 of [Dally and Towles, 2003] for reference). We discussed the basic credit mechanism in Section 3.2.1 before. We refer to the same circuit in this section and reproduce its structure in Figure 5.1 for convenience. It illustrates the basic credit-based flow-control mechanism where a source (S_1) wants to send flits to a sink (S_2) through a buffer (B_2) and the flow of flits across B_2 is controlled by the *credit_logic* sub-circuit. The main purpose of *credit_logic* is to allow only a restricted number of flits to enter the system it is guarding (in this case, the buffer B_2). The particular synchronization achieved by forks and joins in Figure 5.1 gives rise to the following invariant:

$$num(B_1) + num(B_2) = num(B_3)$$

where $num(B)$ denotes the number of flits currently residing in buffer B. We will call this relation BUFFER RELATION in the subsequent sections. While Figure 5.1 is a very simple example of the credit-based flow-control mechanism, the same principle is used in complicated industrial designs. We will consider two such families as case-studies, viz. *virtual channels* and *hardware scoreboards*. These fabrics are often used as micro-architectural idioms in complex hardware systems. We will demonstrate how the high-level knowledge that these circuits use credit-based flow-control can be leveraged to produce a scalable proof of their responsiveness. Section 5.4.1 and Section 5.4.2 are devoted to this discussion. Interestingly, BUFFER RELATION with suitable modifications continues to hold on all such credit-based circuits.

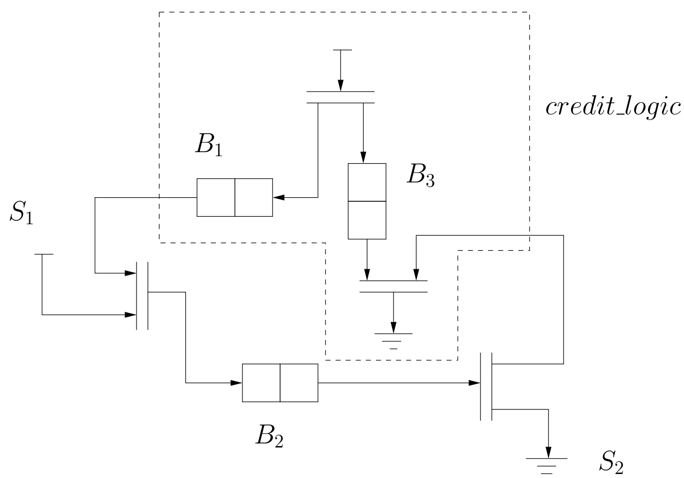


Figure 5.1: Credit mechanism

We mention that these BUFFER RELATIONS, as first identified in [Chatterjee and Kishinevsky, 2010a], are very important for bit-level verification of safety properties of credit-based systems. These seemingly intuitive relations are non-trivial to mine from a bit-level implementation of a fabric, unless they are explicitly hinted by the architects. These relations were used to expedite safety verification in [Chatterjee and Kishinevsky, 2010a]. In the context of response verification, it was not immediately clear how BUFFER RELATIONS can be leveraged. Our main contribution here is to demonstrate how other intermediate invariants can be deduced as corollaries of BUFFER RELATION, eventually leading to a formal proof of response. We devote the whole of Chapter 6 to discussing algorithms that mine BUFFER RELATIONS.

5.3 Response Formulation

In this work, we focus on one particular formal specification of a deadlock property called *response*. It captures an end-to-end progress behavior of the fabrics. The mathematical formulation of the response property of interest is presented in this section. Instead of introducing the formalism in an abstract set up, we illustrate it using an example of a virtual channel as shown in Figure 5.2. [1] It is a reproduction of Figure 3.6, depicting the model $\mathcal{VC}$. See Section 3.2.2 for details on its working principle.

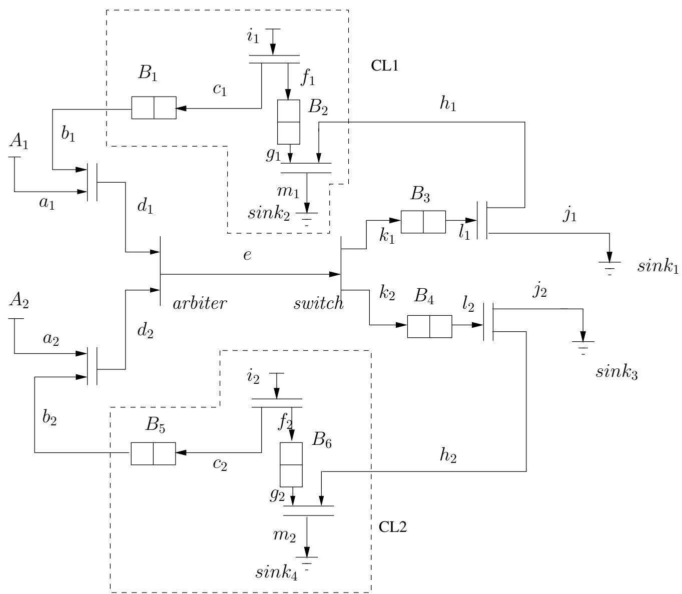


Figure 5.2: Virtual channel

Recall that in the xMAS formalism, each channel c consists of three types of signals, viz. request ($c.req$, a 1-bit signal), grant ($c.gnt$, a 1-bit signal) and data ($c.data$, a bit-vector signal). In Figure 5.2, the sender at source A_1 asserts signal $a_1.req$ when it wants to send a flit. When the network is ready to accept the flit from A_1, it asserts signal $a_1.gnt$. A designer needs to ensure that whenever source A_1 makes a request to send a packet, it will be granted eventually. This ensures that A_1 will never be blocked forever. In LTL, this objective can be written as $\mathbf{G}(a_1.req \Rightarrow \mathbf{F}(a_1.gnt))$. This is a well-known liveness property. While model checking this property, one needs to make *fairness assumptions* on the sinks and on (some of) the sources that they will never cease of work. Otherwise, a trivial buggy scenario can come out where a request from A_1 is never granted because a sink stops draining flits or the credit source in CL1 (or CL2) stops supplying credits. For the virtual channel of Figure

[1]The formulation is uniform across the entire family of fabrics and has no particular relation to the virtual channel design or implementation that we are considering; the same formulation applies to all fabrics considered in this chapter.

5.2, these necessary fairness assumptions may be written in LTL as $\mathbf{GF}(m_1.gnt)$, $\mathbf{GF}(m_2.gnt)$, $\mathbf{GF}(j_1.gnt)$ and $\mathbf{GF}(j_2.gnt)$ for the sinks and $\mathbf{GF}(i_1.req)$ and $\mathbf{GF}(i_2.req)$ for the credit sources. Hence, the overall proof objective $\phi_{response}$ (or ϕ_r in short) is,

$$\phi_r := ((sink\_fair \wedge source\_fair) \Rightarrow response)$$

where

$$\begin{aligned} sink\_fair &:= \mathbf{GF}(m_1.gnt) \wedge \mathbf{GF}(m_2.gnt) \wedge \mathbf{GF}(j_1.gnt) \wedge \mathbf{GF}(j_2.gnt) \\ source\_fair &:= \mathbf{GF}(i_1.req) \wedge \mathbf{GF}(i_2.req) \\ response &:= \mathbf{G}(a_1.req \Rightarrow \mathbf{F}(a_1.gnt)) \end{aligned}$$

The same formulation applies to channel A_2 as well. Note that the above formulation does not refer to $\mathcal{VC}$'s internal signals or topology. Therefore, by careful selection of fairness constraints, the above generic formulation can be easily applied to any other xMAS design.

We will use the terms 'deadlock freedom', 'progress' and 'response' interchangeably in this chapter, though our solution will strictly adhere to the formal specification ϕ_r presented above.

Discussion 1. The above generic formulation is a popular way of expressing a response property, mainly due to the concise way it captures designer's intent. Unfortunately, the price to pay comes from the complexity of the ensuing verification problem. Classical algorithms for verifying the above property would use SCC-oriented approaches, like *nested fixpoint* computation, or *cycle detection*. Both are prohibitively inefficient on real designs. Liveness-to-safety conversion [Schuppan and Biere, 2004], as discussed in the last chapter, is an alternative that promises more scalability. But it is also SCC-oriented and often fails to converge on our experiments with real communication fabrics. As these existing approaches fail to scale on industrially relevant designs, more fine-grained analyses become necessary.

All these existing approaches turn out to be grossly unscalable for our applications. This motivated us to investigate more fine-grained ways of making liveness verification scalable for communication fabrics.

5.4 Approach I : Breaking Into Safety Properties

In this section, we propose a more scalable method for proving response of bit-level implementations of a collection of communication fabrics. This is based on the principle of '*proving liveness using intermediate safety properties*'. We will demonstrate that the end-to-end response property of a fabric can be broken down into a collection of safety properties which are potentially easy-to-verify obligations and once proved on the designs, these safety properties collectively imply the overall response property.

Contributions: Our contributions in this section are summarized as follows:

1. We show how structural and functional safety properties of communication fabrics can be derived and leveraged for verification of their liveness properties. This demonstrates how to capture architect's insights (why the fabric should not deadlock) and to use them to prove response properties via safety model checkers. This approach offers a much more scalable verification solution.

2. Our liveness verification framework is publicly available and works on models written in industry standard languages like Verilog, or standard bit-level description formats like AIGER. We believe that this provides a useful bit-level analysis technique for communication fabrics, where high-level theorem provers and graph theoretic reasoning have been the only proof techniques available so far.

This section is organized in the following sub-sections: Section 5.4.1 presents our methodology of proving liveness properties using intermediate safety invariants on virtual channels. The same methodology is demonstrated on hardware scoreboards in Section 5.4.2. Section 5.4.3 discusses our experiments and Section 5.4.4 concludes the topic with an outline of potential limitations of this approach.

5.4.1 Intermediate Safety Properties for Virtual Channels

We begin with the basic virtual channel $\mathcal{VC}$ shown in Figure 5.2. We will demonstrate how BUFFER RELATIONS can be leveraged to derive auxiliary safety assertions for this design and how satisfaction of these auxiliary safety assertions leads to a proof of ϕ_r. BUFFER RELATIONS for $\mathcal{VC}$ are the following:

$$num(B_1) + num(B_3) = num(B_2)$$

$$num(B_4) + num(B_5) = num(B_6)$$

These relations and a careful analysis of $\mathcal{VC}$ lead to four safety properties as tabulated in Table I as Lemmas 1 through Lemma 4. These properties should hold on the bit-level implementation of the design and a safety verification engine should prove them on the bit-level model. We will discuss our experiments and observations later in Section 5.4.3.

We now introduce another safety property called '*non-blocking property of channel e*' (see Theorem 2 below). It is a non-trivial property, perhaps not apparent from the design immediately and a consequence of carefully architecting the fabric using credit-based flow-control (or to put in other words, it is a consequence of BUFFER RELATIONS). It contributes significantly to the deadlock freedom of $\mathcal{VC}$.

Theorem 2 (Non-blocking e)**.** Whenever $e.req$ is asserted, $e.gnt$ is asserted in the same cycle. In LTL, $\mathbf{G}(e.req \Rightarrow e.gnt)$.

Justification: Suppose $e.req$ is asserted in some cycle. Since it is coming from the arbiter, $e.req$ must correspond to either $d_1.req$ or $d_2.req$. Assume that it corresponds to $d_1.req$, which

Lemma	Description	
Lemma 1	Statement	If both $sink_1$ and $sink_2$ are ready to drain a flit, then either buffer B_2 gets at least one empty slot in the next cycle or $sink_1$ and $sink_2$ persist their states in the next cycle
	in LTL	$j_1.gnt \wedge m_1.gnt \Rightarrow \mathbf{X}(not\_full(B_2) \vee (j_1.gnt \wedge m_1.gnt))$
Lemma 2	Statement	If B_2 has at least one empty slot, it implies that B_1 also has at least one empty slot
	in LTL	$not\_full(B_2) \Rightarrow not\_full(B_1)$
Lemma 3	Statement	For $i = 1, 2$, if B_i has an empty slot and the credit source is not ready to push a flit, the empty slot is preserved in the next cycle
	in LTL	$not\_full(B_i) \wedge !i_1.req \Rightarrow X(not\_full(B_i))$
Lemma 4	Statement	If both B_1 and B_2 have empty slots and the credit source is ready to push a flit, then in the next cycle $b_1.req$ must be asserted
	in LTL	$not\_full(B_1) \wedge not\_full(B_2) \wedge i_1.req \Rightarrow X(b_1.req)$

Table 5.1: Auxiliary safety assertions

implies both $a_1.req$ and $b_1.req$ are asserted in that cycle and buffer B_1 has at least one flit in it. Due to buffer relations $num(B_1) + num(B_3) = num(B_2)$ and $Size(B_2) = Size(B_1) = Size(B_3)$, it is easy to see that buffer B_3 has at least one slot free. Therefore, $k_1.gnt$ must be asserted in that cycle and in turn, $e.gnt$ gets asserted in the same cycle satisfying the property $e.req \Rightarrow e.gnt$. The argument holds for every cycle irrespective of whether the arbiter schedules d_1 or d_2, making $\mathbf{G}(e.req \Rightarrow e.gnt)$ an invariant. □

We will show how this non-blocking property, in association with other safety properties, can lead to a formal proof of ϕ_r. This non-blocking property of a credit-based virtual channel is well-understood by an architect, but has not been leveraged for formal verification of a response property so far. This non-blocking property alone, in spite of being the key behind deadlock freedom of virtual channel, does not immediately lead to a formal proof of ϕ_r. In order to bridge this gap we need additional invariants. We claim that Lemmas 1 through 4 form such a collection of auxiliary safety invariants that are sufficient to achieve a formal proof of satisfaction of ϕ_r starting from Theorem 2. The following theorem justifies this claim.

Theorem 3. If $\mathcal{VC}$ satisfies Theorem 2 and Lemmas 1 through 4, then it also satisfies ϕ_r.

Proof: In order to show that ϕ_r holds on the fabric, it is *sufficient* to show that the signal $b_1.req$ is asserted infinitely often. This is a sufficient condition since whenever $a_1.req$ is asserted and if it is accompanied eventually by an assertion of $b_1.req$, then $d_1.req$ will be asserted. Since the arbiter is fair and every $e.req$ is immediately granted (due to Theorem

2), $d_1.req$ will be scheduled eventually and $d_1.gnt$ will be asserted. Hence, $a_1.gnt$ will be asserted in turn. This shows why every $a_1.gnt$ will be granted eventually if $b_1.req$ is asserted infinitely often.

By the fairness assumptions, gnt signals of $sink_1$ and $sink_2$ will be asserted infinitely often. Suppose the gnt signal of $sink_1$ (i.e. $j_1.gnt$) is asserted in time step t_1 and that of $sink_2$ (i.e. $m_1.gnt$) is asserted in time step t_2. Without loss of generality, assume that $t_1 < t_2$. Now, due to persistence of the signals and the logic of the fork, $j_1.gnt$ will remain asserted until time step t_2. Then due to Lemma 1, one empty slot will be created in B_2 by cycle $t_2 + 1$. By Lemma 2, this creates an empty slot in B_1. By Lemma 3, these empty slots will be preserved if the credit source of CL1 does not push a credit. By fairness of the credit sources, a credit will be pushed eventually and by Lemma 4, this will result in an assertion of $b_1.req$. Due to the nature of the fairness constraints, this chain of events will never cease to work. Hence, $b_1.req$ will be asserted infinitely often. □

5.4.1.1 Variants of Virtual Channel

The same principles of BUFFER RELATION and intermediate safety assertions work for proving ϕ_r for more complex virtual channels as well. For example, consider $\mathcal{VCB}$ and $\mathcal{VCO}$ from Figures 3.7 and 3.8 respectively. We claim that all Lemmas 1 through 4 and Theorem 2 hold on these designs. For Theorem 2, the channel of interest is the one that leaves the arbiter in both Figures 3.7 and 3.8. We claim that theorems analogous to Theorem 2 can be formulated (as presented below) and proved for these complex virtual channels. However, the formulation of the BUFFER RELATIONS will change depending on the structures of the fabrics.

Theorem 4. If $\mathcal{VCB}$ satisfies Theorem 2 and Lemmas 1 through 4, then it also satisfies ϕ_r.

Theorem 5. If $\mathcal{VCO}$ satisfies Theorem 2 and Lemmas 1 through 4, then it also satisfies ϕ_r.

Justifications of these theorems are left to the reader as they are analogous to the case of simple virtual channel. We include verification results of Theorem 2 and Lemmas 1 through 4 for these fabrics in Section 5.4.3.

5.4.1.2 A Virtual Channel That Actually Deadlocks

We conclude the discussion on virtual channels by presenting an example (Figure 5.3) where the virtual channel actually deadlocks and does not satisfy ϕ_r. Interestingly, it does not satisfy the non-blocking property (Theorem 2) and our framework will provide a counter-example to this theorem. The example of the virtual channel shown in Figure 5.2 has a feedback joining the upper fork with the lower source. This feedback channel goes through a function (*flip*) that flips the bit of the flit that determines the output branch of the switch. Note that the upper credit buffer B_2 has three slots. Analysis reveals that this particular buffer size for B_2 is the cause of the violation of ϕ_r. The same fabric with two slots in B_2

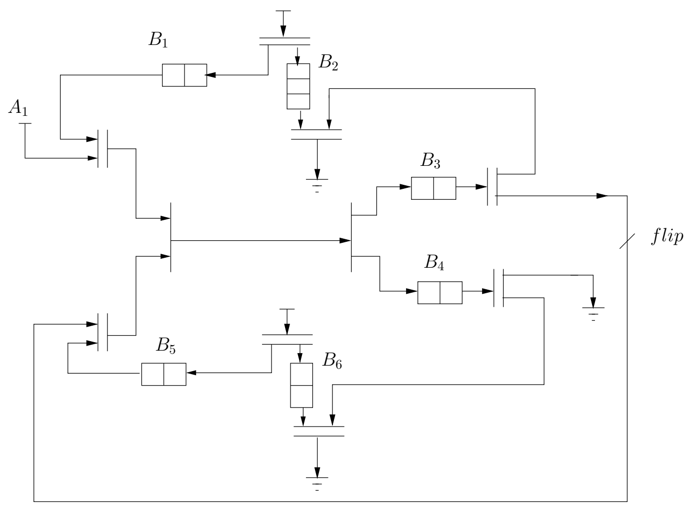


Figure 5.3: A deadlocking virtual channel

works without any deadlock. The fabric $\mathcal{VCO}$ can have a similar scenario if the capacity of B_5 is less than the sum of capacities of B_3 and B_4. Then it would violate ϕ_r. It is quite common to make such mistakes in the bit-level implementations and thereby result in deadlocks.

5.4.2 Hardware Scoreboard

Similar analysis can be performed for the hardware scoreboard from Figure 3.10. Theorem 2 and Lemmas 1 through 4 (with appropriate modifications) can be defined on this structure due to the particular synchronization of credit logic. Theorem 2 is defined on the channel between the join and the switch at the entry of the scoreboard. Lemmas 1 through 4 need to include the role of the arbiter at the exit of the scoreboard. These are only cosmetic changes to the structure of the lemmas. We can again define a theorem analogous to Theorem 3 for the scoreboard and the role of Theorem 2 and Lemmas 1 through 4 in its proof would remain the same even after the necessary cosmetic modifications.

5.4.3 Experimental Results

We developed Verilog implementations of the models of the virtual channels and scoreboards. Our xMAS library implementation closely follows the logical specification provided in [Chatterjee *et al.*, 2010] which makes our benchmarks easily reproducible. We have used ABC [Mishchenko, 2013] as our safety verification engine. Our experiments were performed on a laptop with 1.2 GHz Intel Celeron processor and 2 GB RAM.

We present in Table 5.3 run-times (in sec.) taken by ABC on the various models considered. In all experiments, ABC conjuncted all safety properties (Theorem 2 and Lemma 1 though 4) as a single proof obligation and tried to prove it. We present two sets of experiments. In the first set, we enabled the BUFFER RELATIONS as assumptions and in the second set, these assumptions were not used. From the experimental results, it is clear that BUFFER RELATIONS play a crucial role in verification. For the virtual channel examples, BUFFER RELATIONS made the proof obligation one-step inductive. For the scoreboard example, they sped up the proof. It will be interesting to investigate invariants for scoreboards that would make their proofs one-step inductive too; it is left as a future work. The fact that BUFFER RELATIONS make proofs one-step inductive for many designs is the key factor that makes this approach scalable. The designs that we considered are parameterizable and can serve as components of larger designs. In order to ensure scalability, ideally we need a proof technique whose complexity will be (almost) independent of the design size. One-step induction meets this goal. The BUFFER RELATIONS for other virtual channels and scoreboards used in our proofs are shown in Table 5.2. In those relations, $num_{A_i}(B)$ represents the number of flits in buffer B that came from source A_i, for $i = 1, 2$.

model	BUFFER RELATIONS
buffered vc	$num(B_1) + num(B_3) + num_{A_1}(B_{ch}) = num(B_2)$ $num(B_5) + num(B_4) + num_{A_2}(B_{ch}) = num(B_6)$
ordered vc	$num(B_1) + num(B_3) = num(B_2)$ $num(B_7) + num(B_4) = num(B_8)$ $num(B_3) = num_{A_1}(B_5) + num(B_6)$ $num(B_4) = num_{A_2}(B_5)$ $num(B_5) = num_{A_1}(B_5) + num_{A_2}(B_5)$
scoreboard	$num(B_1) + \sum_{i=5}^{9} num(B_i) = num(B_3)$ $num(B_2) + \sum_{i=10}^{14} num(B_i) = num(B_4)$

Table 5.2: Assumptions used in the proofs

In Table 5.3, a few things may be noted. The number of flip-flops and AND gates in the circuit increases once the BUFFER RELATIONS are put in as assumptions. However from the run-time of ABC, it is evident that this increase in logic only facilitates the proof. In all cases when induction failed, interpolation using the default resource limits of ABC failed as well. However in all cases, property directed reachability (PDR) [Bradley, 2011] succeeded. For the cases where one-step induction succeeded, we have not listed the run-time for PDR. For all cases where the assumptions were supplied, ABC independently proved the assumptions within the time shown in the table. In Figure 3.7, size k of the channel buffer B_{ch} is parameterizable and we only provide run-times for $k = 2$ and 4.

Proof with Assumption					
model	PI	f/f	AND	One step Induction	Other (PDR)
virtual channel (VC)	8	213	4383	0.65	-
buffered VC ($k = 2$)	10	383	8089	1.7	-
buffered VC ($k = 4$)	12	385	8288	1.78	-
ordered VC	13	424	9253	2.26	-
scoreboard	47	1293	18596	failed	38.80
Proof without Assumption					
model	PI	f/f	AND	One step Induction	Other (PDR)
virtual channel	8	211	3449	failed	0.73
buffered VC ($k = 2$)	10	315	4579	failed	99.2
buffered VC ($k = 4$)	12	317	4652	failed	68.81
ordered VC	13	350	5236	failed	26.94
scoreboard	47	1289	14942	failed	57.25

Table 5.3: Experiment on various communication fabrics

5.4.4 Conclusion and Limitations

We presented a way of verifying response property of credit-based flow-control networks by deriving and proving intermediate safety properties. These were derived by leveraging the high-level structure of the network. A crucial set of invariants called BUFFER RELATIONS were derived. These relations made the proof one-step inductive for many examples and sped up the proof for others. We believe this work exposes new opportunities of research in formal verification of communication fabrics. The current work-flow of our framework is to prove Theorem 2 and Lemma 1-4 automatically using a safety verification engine and then take refuge in Theorem 3 to conclude the satisfaction of ϕ_r. This leads to the question whether we can automate the proof of Theorem 3 as well; this is left as a future work. Also, necessity and adequacy issues of Lemmas 1 through 4 are yet to be studied. One important question is whether these safety properties can be mined automatically and whether they are effective for proving other kinds of progress properties. Finally, it needs to be studied whether the approach presented could be utilized to prove a response property of a chip-wide communication network. Compositional reasoning, along with our approach, may prove to be useful in this context.

5.5 Approach II: Well-founded Induction

The procedure of breaking down a liveness property into intermediate safety properties, as presented in the previous section, poses two challenges. First, we need to derive the intermediate safety properties and prove them on the design. Then we need to establish that the proof of all these safety properties implies the target liveness property. In the last section, Theorem 3 was devised to achieve the last goal. But these kinds of theorems are complicated to devise and prove. In this section, we will demonstrate how the notion of well-founded induction can simplify this task.

Theorems like Theorem 3 are rather ad-hoc in nature, but well-founded induction is a structured way of reasoning about liveness. In fact, the classic framework of decomposing liveness into intermediate safety properties, as proposed by Manna and Pnueli in the 1980's [Manna and Pnueli, 2010], relies on the notions of well-founded induction and ranking functions. It is only very recently that this approach has been applied successfully to liveness verification of non-trivial systems of practical importance. For example, success was reported by Cook et al. in proving termination and other liveness properties of system codes using the principle of well-founded induction [Cook *et al.*, 2006], [Cook *et al.*, 2007]. While this approach seems to be promising way of proving liveness of complex systems efficiently, the main hurdle lies in discovering the well-founded order hidden in the target system. In this section, we address this challenge in the context of communication fabrics. We show that it is possible to discover a well-founded order in reactive behaviors of communication fabrics and that it leads to efficient verification of their liveness properties. We focus on the same family of communication fabrics (i.e. credit-based flow-control systems) and the same response property described in Section 5.3.

Our contributions in this section are summarized as follows:

- Our analyses of a group of credit-based flow-control systems expose well-founded orders hidden in these designs. We show how these relations can simplify the liveness proof process.

- We demonstrate rigorously how fairness assumptions interact with a system to ensure its liveness. This offers insight into the mechanisms for satisfaction of liveness properties for our target systems. Current liveness verification algorithms implemented in the bit-level verification systems are not customized for this kind of mechanism. We hope that this observation will be useful for designing more efficient and scalable liveness verification tools in the future.

- Our well-founded order analyses demonstrate that the systems we considered constitute a family of designs whose response verification problem is easier to solve compared to the complexity of the problem in its fully general form. In the future, this may lead to a possible hierarchy of problems based on how hard it is to solve their response verification. But this is outside the scope of this dissertation and left as future work.

Below in Section 5.5.1, we discuss how well-founded order works on different variants of virtual channels and in the following section, we extend the same idea to more complex systems. Section 5.7.5 presents our experimental results.

5.5.1 Well-Founded Structure for $\mathcal{VC}$

A careful study of the RTL description of $\mathcal{VC}$ reveals the following logical relation:

$$a_1.gnt = a_1.req \land b_1.req \land k_1.gnt \land u$$

where u is the state element of the arbiter as mentioned in Section 3.1. Informally, it means that when $a_1.req$ is asserted by source A_1, $a_1.gnt$ will be available only when all of the following three conditions are satisfied simultaneously: (*i*) buffer B_1 is not empty, (*ii*) buffer B_3 is not full and (*iii*) the arbiter has parked its grant to source A_1. When $a_1.req$ arrives, if any one of these three conditions is not met, $a_1.gnt$ will be deferred until all three are satisfied together (due to persistence property, $a_1.req$ will remain asserted until $a_1.gnt$ is received). Our goal is to show that the basic virtual channel admits a well-founded relation which guarantees that even if any one of the three conditions is not met when $a_1.req$ arrives, all three will be eventually satisfied together at some point in time resulting in an eventual assertion of $a_1.gnt$. Figure 5.4 shows a diagrammatic representation of this well-founded relation. We call this diagrammatic representation a *well-founded structure* (WFS). A high-level explanation of this structure is presented next, followed by a detailed description including its role in response verification.

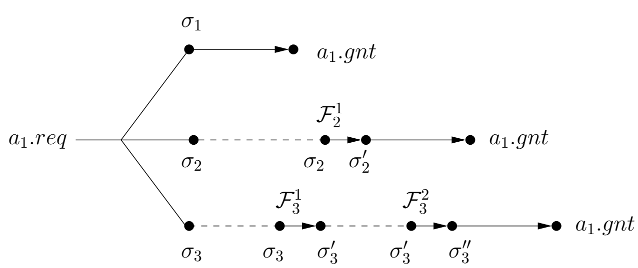


Figure 5.4: Well-founded structure for virtual channel

5.5.1.1 High-level Explanation

Figure 5.4 depicts that the basic virtual channel $\mathcal{VC}$ can be in any one of the three configurations $\{\sigma_1, \sigma_2, \sigma_3\}$ when $a_1.req$ arrives. Depending on which configuration it is in, it will wait for different rounds of *fairness events* $\mathcal{F}_i^j$. In response to each such fairness event, the system will make 'step-by-step' progress toward the target configuration which satisfies $a_1.gnt$. The horizontal trajectories starting from a configuration σ_i and ending at $a_1.gnt$ represent various paths along which the system might evolve. $\mathcal{F}_2^1, \mathcal{F}_3^1, \mathcal{F}_3^2 \subseteq \mathcal{F}$ denote the necessary fairness events where $\mathcal{F} = \{i_1.req, i_2.req, j_1.gnt, j_2.gnt, m_1.gnt, m_2.gnt\}$ is the set of all fairness signals associated with the model. In our notation, a fairness event $\mathcal{F}_i^j = \{f_1, f_2, \ldots, f_n\} \subseteq \mathcal{F}$ essentially represents the conjunction $f_1 \wedge f_2 \wedge \ldots \wedge f_n$. We will use the set notation and the conjunction notation interchangeably for fairness events. The main idea behind Figure 5.4 is that whichever path the system might evolve from the point of receiving $a_1.req$, it will eventually hit $a_1.gnt$ and this is guaranteed by its 'step-by-step' progress under the rounds of fairness events. In Figure 5.4, a configuration σ_k with a superscript $\mathcal{F}_i^j$ followed by another configuration σ_k' denotes that the model stutters at σ_k and only upon the arrival of fairness event $\mathcal{F}_i^j$ makes a transition to σ_k'.

5.5.1.2 Detailed Explanation

A *configuration* σ is a state predicate which represents one or more concrete states of the system. $\mathcal{VC}$ can be in any one of the three mutually disjoint configurations when $a_1.req$ arrives. These configurations are represented by σ_1, σ_2 and σ_3 with the following definitions:

$$\begin{aligned}
\sigma_1 &= \neg Empty(B_1) \wedge \neg Full(B_3) \\
\sigma_2 &= Empty(B_1) \wedge \neg Full(B_3) \\
\sigma_3 &= Empty(B_1) \wedge Full(B_3)
\end{aligned}$$

$Empty(B)$ and $Full(B)$ are simple predicates defined on buffer B with the following obvious semantics: $Empty(B) = 1$ iff buffer B contains no flit and $Full(B) = 1$ iff buffer B has no slot free for any more flits. In terms of the RTL signals, $Empty(B_1) \triangleq \neg b_1.req$ and $Full(B_3) \triangleq \neg k_1.gnt$. Note that there is no configuration $\sigma_4 = \neg Empty(B_1) \wedge Full(B_3)$ that the model can assume when $a_1.req$ arrives. This is an impossible scenario due to the BUFFER RELATION $num(B_1) + num(B_3) = num(B_2)$ which precludes a situation where buffer B_1 and B_3 together have more than 2 flits (as stipulated by σ_4). Hence, $\sigma_1 \vee \sigma_2 \vee \sigma_3$ exhaustively captures all possible scenarios.

Discussion 2. The above situation is an obvious one where we infer a (safety) property of the system as a corollary of some of its known safety properties. It will influence the well-founded structure as well as the overall proof of liveness. We will encounter similar situations recurrently in subsequent arguments. To highlight this phenomenon where safety properties influence satisfaction of liveness, we will mark the relevant corollaries with a superscript symbol $^{[\text{safe}]}$. □

From the above discussion, we see that three predicates σ_1, σ_2 and σ_3 form mutually disjoint but exhaustive partition of the collection of possible scenarios. This justifies the three-way branching in Figure 5.4. We use solid lines to capture this case-split. We use two more types of lines in a WFS, *viz.* broken lines and thick solid arrows. Broken lines denote eventuality, *i.e.* passage of an indefinite number of time steps and thick solid arrows denote passage of one or more, but a definite number of time steps. With these semantics of lines in mind, we now focus on the individual branches of the structure of Figure 5.4. At the end of our analysis, we will collect a set of safety properties of the form φ_i^j or ψ_k^l (explained later) that will substitute for the need of a traditional liveness proof of ϕ_r.

[**Case 1:** σ_1 **to** $a_1.gnt$] If the arbiter is ready to grant A_1 (*i.e.* $u = 1$), then due to the non-blocking property of e, $a_1.gnt$ is received in the same cycle [safe]. Otherwise, if the arbiter is offering the grant to A_2, again due to round-robin nature of the arbiter and the non-blocking property of e, the arbiter will turn to A_1 in the immediately next cycle [safe]. Hence $a_1.gnt$ will be received within at most one cycle (in this context, note the use of thick solid arrow in Figure 5.4). Therefore, we can check if a bit-level implementation of $\mathcal{VC}$ indeed behaves in this way under **Case 1** by formally verifying the following safety property on the bit-level implementation:

$$\varphi_1^1 \triangleq a_1.req \wedge \sigma_1 \Rightarrow a_1.gnt \vee \mathbf{X}(a_1.gnt)$$

[**Case 2 :** σ_2 **to** $a_1.gnt$] Note the following corollary: [safe]

$$\psi_2^1 \triangleq Empty(B_1) \wedge \neg Full(B_3) \Rightarrow \neg Full(B_2)$$

Therefore, we need to wait for the credit source i_1 to push a flit in buffer B_1. As per our fairness assumption $\mathbf{GF}(i_1.req)$, we may need to wait to get $i_1.req$, but its eventual arrival is always guaranteed. This arrival is denoted by the fairness event $\mathcal{F}_2^1 \triangleq \{i_1.req\}$. Once $\mathcal{F}_2^1$ happens when the system is in σ_2, it moves to configuration σ_2'. Incidentally, σ_2' is same as σ_1. Hence the analysis of **Case 1** follows. In order to check if a bit-level implementation behaves in this way, we need to check the following safety properties φ_2^1 and φ_2^2 in addition to φ_1^1 and ψ_2^1.

$$\varphi_2^1 \triangleq \sigma_2 \wedge \neg\mathcal{F}_2^1 \Rightarrow \mathbf{X}(\sigma_2), \quad \varphi_2^2 \triangleq \sigma_2 \wedge \mathcal{F}_2^1 \Rightarrow \mathbf{X}(\sigma_1)$$

[**Case 3 :** σ_3 **to** $a_1.gnt$] Note the following corollary: [safe]

$$\psi_3^1 \triangleq Empty(B_1) \wedge Full(B_3) \Rightarrow Full(B_2)$$

Therefore, at least one empty slot needs to be created in buffer B_2 before credit source i_1 can push a flit into B_1 (as well as in B_2). For this, we need the fairness event $\mathcal{F}_3^1 \triangleq \{j_1.gnt, m_1.gnt\}$ to occur to remove a flit from each of B_2 and B_3. Then we need the fairness event $\mathcal{F}_3^2 \triangleq \{i_1.req\}$ to push a flit into buffer B_1. The system, initially stuttering on σ_3, jumps to $\sigma_3' = \sigma_2$ upon arrival of $\mathcal{F}_3^1$. Henceforth from σ_3', it follows the analysis of **Case 2**. In order to check if a bit-level implementation satisfies these additional behaviors under **Case 3**, we need to check, in addition to $\varphi_1^1 \wedge \varphi_2^1 \wedge \varphi_2^2 \wedge \psi_2^1 \wedge \psi_3^1$, the following safety properties.

$$\varphi_3^1 \triangleq \sigma_3 \wedge \neg\mathcal{F}_3^1 \Rightarrow \mathbf{X}(\sigma_3), \quad \varphi_3^2 \triangleq \sigma_3 \wedge \mathcal{F}_3^1 \Rightarrow \mathbf{X}(\sigma_2)$$

Discussion 3. The final outcome of the well-foundedness analysis of a communication fabric is a set of safety properties which ensures the validity of liveness of the model. In case of the basic virtual channel, we derive the set of properties $\mathbb{S} = \{\sigma_{exhaust}\} \cup \{\varphi_1^1, \varphi_2^1, \varphi_2^2, \varphi_3^1, \varphi_3^2\} \cup \{\psi_2^1, \psi_3^1\}$. $\sigma_{exhaust} = a_1.req \Rightarrow \sigma_1 \vee \sigma_2 \vee \sigma_3$ captures all the configurations that the system can assume when $a_1.req$ arrives. While ψ_i^j are design invariants that restrict the state space of the system, φ_i^j are invariants that are more closely related to the progress of the system. Proof of the properties in $\mathbb{S}$ indirectly testify the validity of ϕ_r. Therefore, an efficient proof of the properties in $\mathbb{S}$ may replace a separate rather inefficient proof attempt for ϕ_r. See Section 5.7.5 for the relevant experimental results. □

5.5.1.3 Virtual Channel with Order $\mathcal{VCO}$

We apply the same principle to the more complex structure $\mathcal{VCO}$ (Figure 3.8), and obtain a WFS as shown in Figure 5.5.

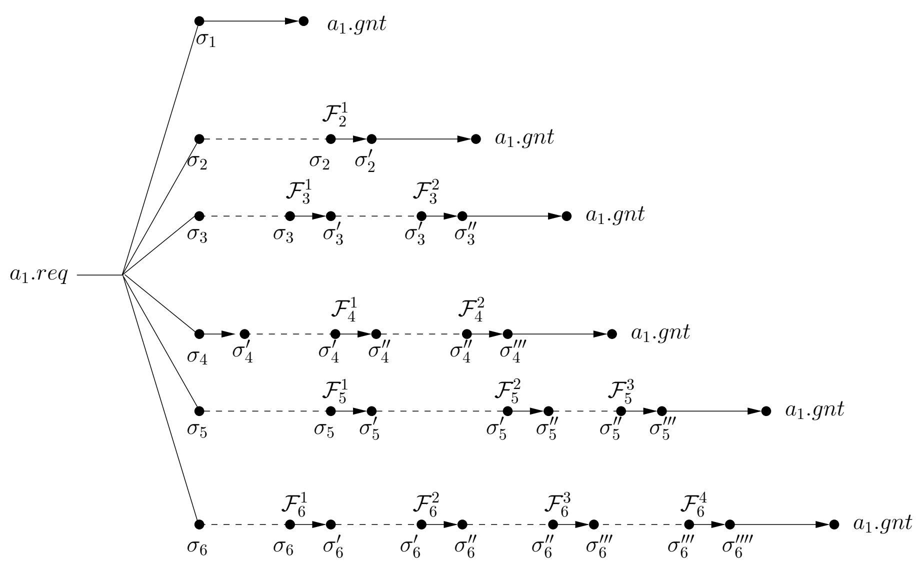


Figure 5.5: WFS for virtual channel with order

The WFS of Figure 5.5 contains some intriguing scenarios which were not observed in the WFS for the basic virtual channel. Here $\{\sigma_i : i = 1, 2, \ldots, 6\}$ is the set of mutually disjoint configurations that the model can assume when $a_1.req$ arrives. Definitions of σ_i's are the

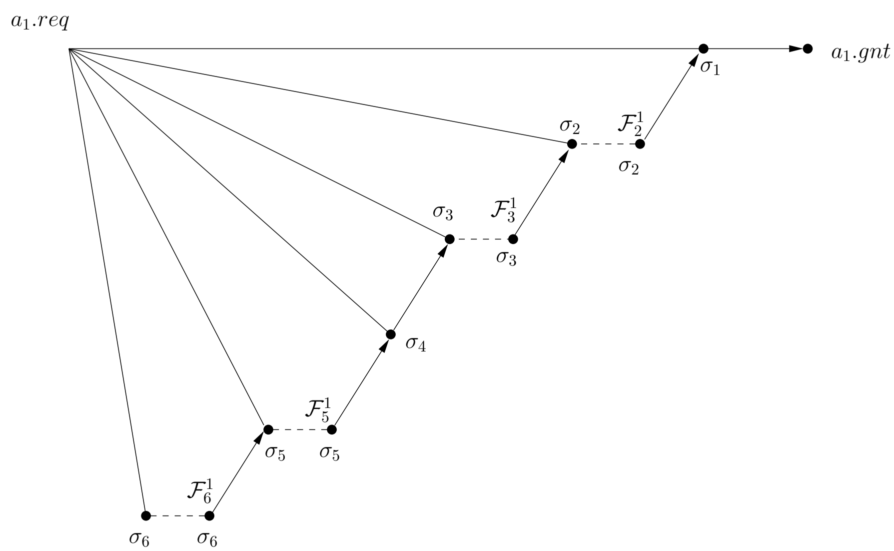


Figure 5.6: A folded representation the same

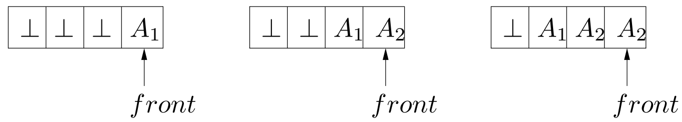
following:

$$
\begin{aligned}
\sigma_1 &= \neg Empty(B_1) \\
\sigma_2 &= Empty(B_1) \wedge \neg Full(B_3) \\
\sigma_3 &= Empty(B_1) \wedge Full(B_3) \wedge \neg Empty(B_6) \\
\sigma_4 &= Empty(B_1) \wedge Full(B_3) \wedge Empty(B_6) \wedge \pi_0(B_5) \\
\sigma_5 &= Empty(B_1) \wedge Full(B_3) \wedge Empty(B_6) \wedge \pi_1(B_5) \\
\sigma_6 &= Empty(B_1) \wedge Full(B_3) \wedge Empty(B_6) \wedge \pi_2(B_5)
\end{aligned}
$$

While $Empty(B_i)$ and $Full(B_i)$ are predicates as defined before, $\pi_0(B_i)$, $\pi_1(B_i)$ and $\pi_2(B_i)$ are three new, rather non-trivial, situation-specific predicates defined as follows:

- $\pi_0(B_i) = 1$ iff the flit at the front of the buffer B_i is coming from source A_1,
- $\pi_1(B_i) = 1$ iff the flit at the front of the buffer B_i is coming from source A_2 and the flit at the second position from the front is coming from A_1,
- $\pi_2(B_i) = 1$ iff two consecutive flits at the front of the buffer B_i are coming from source A_2 and the third flit is coming from source A_1.

The situations captured in π_0, π_1 and π_2 are presented in Figure 5.7. Note that the buffer slots which are ignored by each of π_i's are marked with $\perp$ (for don't care). The significance of these predicates is discussed in the following sections.



(a) $\pi_0(B_5)$ (b) $\pi_1(B_5)$ (c) $\pi_2(B_5)$

Figure 5.7: Various configurations of B_5 (captured in predicates π_1, π_1, π_2)

Each of the paths of Figure 5.6 eventually leads to a configuration where $a_1.gnt$ holds. Every such path to $a_1.gnt$ depends on a particular sequence of fairness events which are defined below:

$\mathcal{F} = \{i_1.req, i_2.req, j_1.gnt, j_2.gnt, j_3.gnt, m_1.gnt, m_2.gnt\}$

$\mathcal{F}_2^1 = \{i_1.req\}, \mathcal{F}_3^1 = \{j_1.gnt, m_1.gnt\}, \mathcal{F}_3^2 = \{i_1.req\}$
$\mathcal{F}_4^1 = \{j_1.gnt, m_1.gnt\}, \mathcal{F}_4^2 = \{i_1.req\}$
$\mathcal{F}_5^1 = \{j_2.gnt, j_3.gnt, m_2.gnt\}$
$\mathcal{F}_5^2 = \{j_1.gnt, m_1.gnt\}, \mathcal{F}_5^3 = \{i_1.req\}$
$\mathcal{F}_6^1 = \mathcal{F}_6^2 = \{j_2.gnt, j_3.gnt, m_2.gnt\}$
$\mathcal{F}_6^3 = \{j_1.gnt, m_1.gnt\}, \mathcal{F}_6^4 = \{i_1.req\}$

While analyses of the paths from configurations σ_1, σ_2 and σ_3 are very similar to the three paths shown in Figure 5.4, analyses of the paths from configurations σ_4, σ_5 and σ_6 require special attention. All three configurations σ_4, σ_5 and σ_6 correspond to the situation where buffer B_1 is empty, B_3 is full and B_6 is empty, but they differ in the configuration of buffer B_5 (in particular, the ordering of different *types* of flits in B_5). Among various ordering of flits in B_5, three relevant ones are captured in π_0 (corresponding to configuration σ_4), π_1 (corresponding to σ_5) and π_2 (corresponding to σ_6) as already defined. Configuration σ_4 immediately jumps to configuration σ_3 in the next time step ([safe]), hence it follows the same analysis of configuration σ_3. Under configuration σ_5, we first need to remove the top-most A_2-flit from B_5 and for this we need to wait for fairness event $\mathcal{F}_5^1$. Then it becomes the same as configuration σ_4 and the same analysis follows. Similarly, under configuration σ_6, we need to wait for two consecutive fairness events $\mathcal{F}_6^1$ and $\mathcal{F}_6^2$ to get an A_1-flit at the top. The analysis for configuration σ_4 follows (equivalently, we can say that configuration σ_6 jumps to σ_5 upon arrival of fairness event $\mathcal{F}_6^1$ and follows the analysis of σ_5 thereafter).

As before, we derive the set of safety properties $\mathbb{S} = \{\sigma_{exhaust}\} \cup \{\varphi_i^j\} \cup \{\psi_k\}$ where

$\sigma_{exhaust} = \bigvee_{i=1}^{6} \sigma_i$ and $\{\varphi_i^j\}$ and $\{\psi_k\}$ contain the following properties:

$$\begin{array}{rcl}
\varphi_1^1 & \triangleq & a_1.req \wedge \sigma_1 \Rightarrow a_1.gnt \vee \mathsf{X}(a_1.gnt) \\
\varphi_4^1 & \triangleq & \sigma_4 \Rightarrow \mathsf{X}(\sigma_3) \\
\varphi_i^1 & \triangleq & \sigma_i \wedge \neg\mathcal{F}_i^1 \Rightarrow \mathsf{X}(\sigma_i), \quad \forall i \in \{2,3,5,6\} \\
\varphi_i^2 & \triangleq & \sigma_i \wedge \neg\mathcal{F}_i^1 \Rightarrow \mathsf{X}(\sigma_{i-1}), \quad \forall i \in \{2,3,5,6\} \\
\psi_1 & \triangleq & num(B_1) + num(B_3) = num(B_2) \\
\psi_2 & \triangleq & num(B_7) + num(B_4) = num(B_8) \\
\psi_3 & \triangleq & num(B_3) = num(B_6) + num_{A_1}(B_5) \\
\psi_4 & \triangleq & num(B_4) = num_{A_2}(B_5) \\
\psi_5 & \triangleq & num(B_5) = num_{A_1}(B_5) + num_{A_2}(B_5)
\end{array}$$

where $num_{A_1}(B)$ and $num_{A_2}(B)$ represent the number of flits in buffer B that come from source A_1 and A_2 respectively.

Discussion 4. The path that follows from σ_6 to $a_1.gnt$ is particularly interesting. It illustrates how the ordering logic of this virtual channel can complicate the proof of liveness. The path requires two consecutive fairness events $\mathcal{F}_6^1$ and $\mathcal{F}_6^2$, which are essentially the same event. In spite of their identity, the configurations σ_6' and σ_6'', which are assumed by VCO upon arrival of these events respectively, are different. In terms of progress, σ_6'' should be considered closer to the target state $a_1.gnt$ than σ_6'. Note that if the buffer sizes of the fabric change, we may need more rounds of the same fairness events. Unfortunately, this progress-ensuring insight is missing in the monolithic formulation of ϕ_r. A traditional liveness verifier may need to learn this design behavior itself in an unguided way before it derives a proof for ϕ_r. We suspect that this is the reason why a traditional model checker finds it challenging to prove ϕ_r on a fabric with less than hundred flip-flops (see Section 5.7.5 for run-times). □

Discussion 5. Note that the ordering logic makes the analyses of well-foundedness of the response property for source A_1 and A_2 asymmetric. The well-founded structures for the response property for source A_1 and A_2 are the same, with the same event orderings (only difference lies in the actual definitions of the fairness events $\mathcal{F}_i^j$'s) for the basic virtual channel and the virtual channel with channel buffer. But for the virtual channel with ordering, the structures will be different for A_1 and A_2. The structure for source A_2 would be simpler and similar to that of Figure 5.4. We have analyzed the case of A_1, which is more complex between the two and leave the simpler one for the reader as an exercise. □

Discussion 6. As discussed, one path from a configuration σ_i to $a_1.gnt$ can use another path from some other configuration σ_j as a subpath, both in the case of the basic virtual channel and the virtual channel with order. This is due to various identities of intermediate configurations (of the form $\sigma_i' = \sigma_j$) and identities of the fairness events (*eg.* $\mathcal{F}_5^2 = \mathcal{F}_4^1$ for virtual channel with order). This allows folding the WFS into a more compact structure as shown in Figure

5.6. It is the folded version of the WFS shown in Figure 5.5 with the same semantics for various lines. It demonstrates an interesting monotonicity among the various cases captured in σ_i's. We can draw a similar folded WFS for the basic virtual channel as well. □

5.5.1.4 Virtual Channel with Channel Buffer

Figure 3.7 shows another variant of the basic virtual channel. It has a buffer (B_{ch}) of size k along the shared channel and can be seen in network-on-chip applications. The particular instance shown in the figure has a channel buffer with $k = 4$. In spite of having a different set of design invariants due to the mixing of two types of flits in B_{ch}, this virtual channel yields the same well-founded structure as in Figure 5.4 with the same definitions for $\mathcal{F}$ and $\mathcal{F}_i^j$'s. The initial configurations σ_1, σ_2, and σ_3 will have minor differences in their formulations because the presence of B_{ch} leads to a different set of design invariants. The new configurations are[2]:

$$\begin{aligned}
\sigma_1 &= \neg Empty(B_1) \wedge \neg Full(B_2) \\
\sigma_2 &= Empty(B_1) \wedge \neg Full(B_2) \\
\sigma_3 &= Empty(B_1) \wedge Full(B_2)
\end{aligned}$$

Careful study shows that the structure is rather insensitive to the size of the channel buffer, provided that the capacity of B_3 and B_4 are large enough to store the credit flits currently circulating in the network. However, a traditional model checker may not be able to leverage these insights automatically and may be choked even for a moderate size of the channel buffers. See Section 5.7.5 for further details.

5.5.2 Complex Systems and Composition of WFS

Well-foundedness may be used for proving the response property of larger systems built on the credit-based flow-control principle of virtual channels. We consider a family of master/slave designs which have an overall similar topology, but they differ in the details of their connectivities and operations. These details often make analysis of one much harder than another. In Figure 3.9, we consider a simple master/slave design that uses virtual channels for deadlock prevention. Note the structural back-to-back composition of two buffered virtual channels of Figure 3.7 in this model and the acyclicity of data-flow in it. Leveraging these two features we may compose the WFS of two buffered virtual channels together and derive the same for the basic master/slave model. The resulting WFS is shown in Figure 5.8 in its folded form. Here, the configurations have the following definitions:
$\sigma_1 = \neg E(B_1)$, $\sigma_2 = E(B_1) \wedge \neg E(B_3)$,
$\sigma_3 = E(B_1) \wedge E(B_3) \wedge \neg F(B_6)$

[2]It is interesting to note that this set of configurations can be used for the basic virtual channel as well. In fact, this set is equivalent to the set used in Section 5.5.1 for the basic virtual channel, but not for the virtual channel with channel buffer

$\sigma_{4.1} = E(B_1) \wedge E(B_3) \wedge F(B_6) \wedge \neg E(C_1)$
$\sigma_{4.2} = E(B_1) \wedge E(B_3) \wedge F(B_6) \wedge E(C_1) \wedge \neg E(C_3)$
$\sigma_{4.3} = E(B_1) \wedge E(B_3) \wedge F(B_6) \wedge E(C_1) \wedge E(C_3) \wedge \neg F(C_6)$
$\sigma_{4.4} = E(B_1) \wedge E(B_3) \wedge F(B_6) \wedge E(C_1) \wedge E(C_3) \wedge F(C_6)$

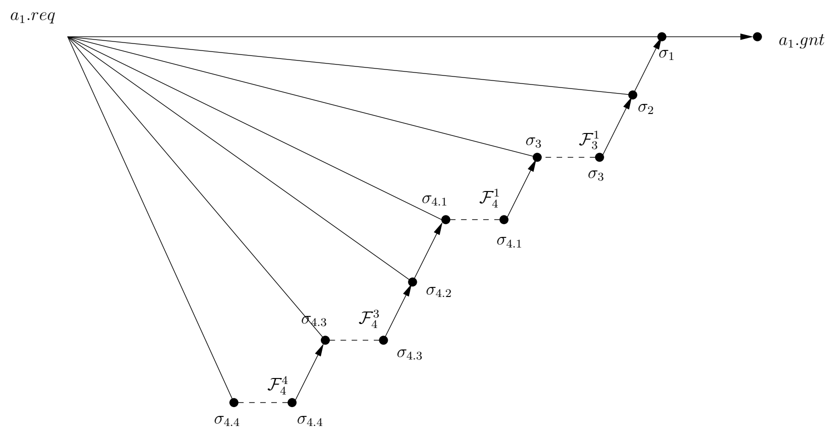


Figure 5.8: Well-founded structure for Master/Slave design

5.5.2.1 Assume/Guarantee Reasoning

Figure 5.9 shows another approach to prove the response property for the master/slave design. It leverages the acyclicity of the data-flow further and uses an assume/guarantee paradigm to break the system into two parts along the break-line of Figure 5.9. This breaks the overall proof obligation ϕ_r into two sub-obligations. Along the break-line, we introduce new sinks and sources (highlighted by dark circles). It is assumed that a sink is fair, which is guaranteed by proving another response property on the source. The advantage of this approach is that we do not have to compose the WFS of two sub-blocks. A disadvantage is that most of the networks may not be broken into parts so easily. For example, we considered some other master/slave designs not easily decomposed and composition of WFS seemed to be the only technique that we could use.

5.5.3 Experimental Results

We ran two sets of experiments: in one, we attempted to prove the native liveness property ϕ_r without the help of the safety invariants. In the other experiment, we attempted to prove the safety properties in $\mathbb{S}$ which aims to replace ϕ_r. For the first set, we used the liveness-to-safety transformation to convert the liveness verification problem to an equi-satisfiable safety verification problem (see Chapter 4 for details). In both experiments, we use interpolation,

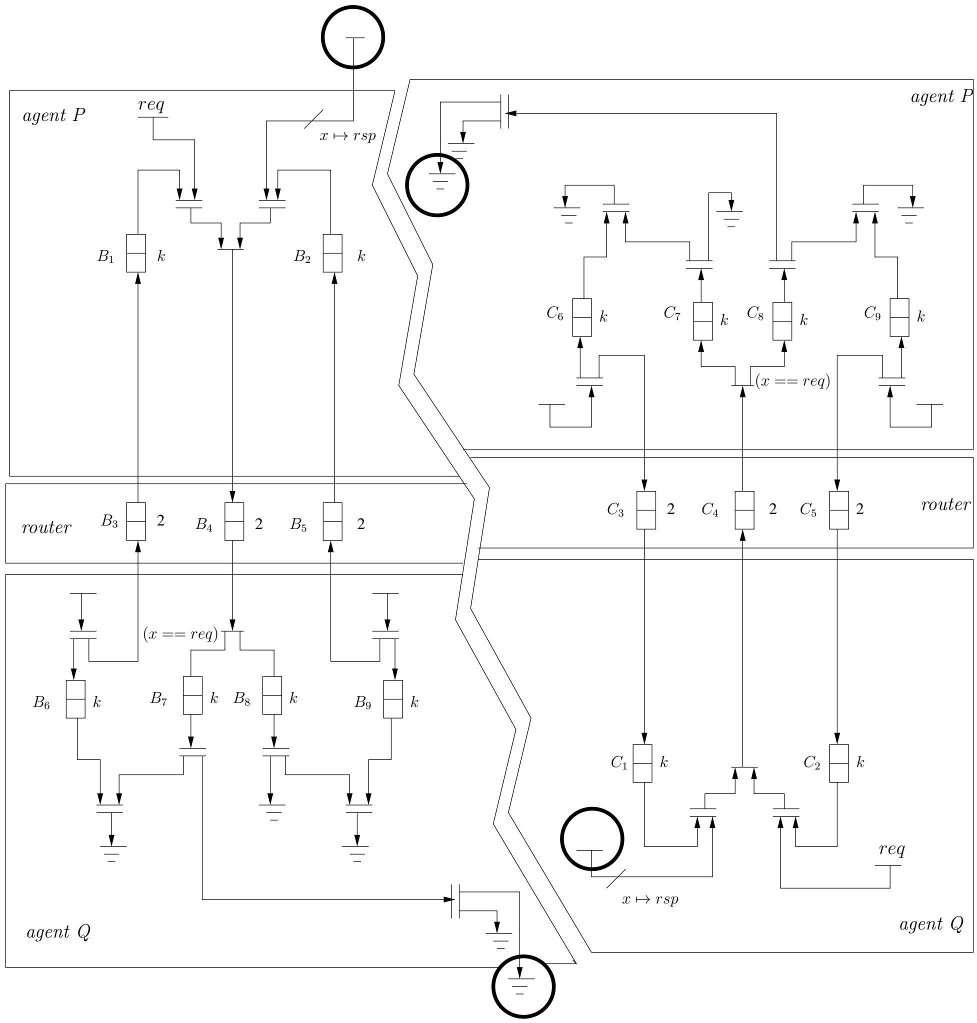


Figure 5.9: Compositional structure for Master/Slave design

property directed reachability (PDR) and induction as safety proof engines on the final single-output safety property instance. Verification run-times of these experiments are shown in Table 5.4. A '-' symbol in place of run-time means the corresponding engine could not prove the property within the default resource limit of ABC. In Table 5.4, we present run-times for the basic virtual channel, virtual channel with channel buffer, virtual channel with ordering and a few cases where several instances of virtual channels with ordering are coupled in cascade (Vco$i$ stands for i instances of Virtual Channel with ordering are cascaded).

Native Liveness					
Design	#PI	#flop	proving ϕ_r after L2S		
			Interpolation	PDR	Induction
Virtual Channel	8	59	1.52	12.78	-
Virtual Channel with Buffer	8	77	-	1515.20	-
Virtual Channel with Order	9	104	-	126.44	-
Vco2	14	205	-	-	-
Vco3	19	278	-	-	-
Vco4	24	380	-	-	-
Liveness With Intermediate Safety					
Design	#PI	#flop	proving properties in $\mathbb{S}$		
			Interpolation	PDR	Induction
Virtual Channel	8	59	0.17	0.13	0.03
Virtual Channel with Buffer	8	77	-	80.95	0.08
Virtual Channel with Order	9	104	-	3.91	-
Vco2	14	205	-	3.91	-
Vco3	19	278	-	54.80	-
Vco4	24	380	-	304.62	-

Table 5.4: Verification run-times for intermediate assertions on well-foundedness

5.6 Approach III : Skeleton Independent Proof Heuristics

In this approach, we show that a very natural subset of design signals of communication fabrics may be treated as candidate hints and, given such a set of candidate hints, we demonstrate how we can algorithmically construct a proof of a response property. Our contributions in this section are the following:

- We propose a methodology and associated algorithms for proving response properties of communication fabrics using a ranking-oriented approach. We make our algorithm efficient and scalable by using a heuristic called *skeleton independent proof* [Bradley *et al.*, 2011]. The resulting algorithm is sound, and complete relative to a set of designer provided hints. (Section 5.6.2, 5.6.3)

- We provide a novel characterization of the state-spaces of communication fabrics and subsequently use it in our algorithm. We hope that this characterization will foster further research, and promote design of even more powerful algorithms. (Section 5.6.2)

- Our algorithm can output a certificate of liveness for correct systems which can be presented to a designer who should be able to analyze the certificate without much effort, and conclude if the system is behaving in the intended way. No temporal logic model checking background is necessary for the inspection of these certificates. (Section 5.6.2)

- In our proof scheme, fairness signals are interpreted more from a designer's perspective, rather than from the traditional perspective of automata-theoretic model checking.
- With reference to the *temporal hierarchy* [Manna and Pnueli, 1991], we extend the scope of the *skeleton independent proof* technique by applying it to a *recurrence* property, compared to a *guarantee* property on which the technique was originally applied. Further, our benchmarks are more complex, both in size and in design complexity, than the simple counter that was used in [Bradley *et al.*, 2011]. This supports the strength and effectiveness of this new heuristic for solving designs with more complex dynamics.

This section is organized in the following sub-sections: we overview terminologies and notations next. In Section 5.6.2, we discuss our characterization of the state-space, the specialized algorithms for ranking-oriented proof, and their efficient implementations using system specific heuristics. In Section 5.6.3, we illustrate the concepts presented in Section 5.6.2 using our running example $\mathcal{VC}$. Section 5.6.4 presents experimental results, and Section 5.6.5 summarizes and highlights possible future work.

5.6.1 Terminologies and Notations

5.6.1.1 Pending Graph, Goal Region, Pending Region

In the verification of a response property, it is helpful to focus on the ***pending graph*** of the design which is defined as follows: The ***goal region*** of the state-space is a subset of states of the state transition graph which can issue a grant without waiting for an external event or input. For example, the goal region of the state-space of $\mathcal{VC}$ is the subset of states where buffer B_1 has at least one flit, *and* buffer B_3 has at least one empty slot, *and* the arbiter is offering the grant to side A_1. The ***pending region***, on the other hand, is the remaining part of the state-space which cannot issue a grant itself, but needs to transit to the goal region (eventually) with the help of external events. The pending graph is obtained by pruning all transitions coming out of the goal region. If we can prove that any state in the pending region cannot reach itself infinitely often without visiting the goal region, this essentially proves that the given design satisfies the response property. It should be noted that in our proof algorithms, pending graphs will not be constructed explicitly; rather the algorithm will work symbolically with the And-Inverter graph (AIG) representation of the fabric models.

5.6.1.2 Notations

Let $\mathcal{F} : (\bar{x}, \bar{i}, \text{init}(\bar{x}), T(\bar{x}, \bar{i}, \bar{x}'))$ denote the *finite state system* underlying to a communication fabric where $\bar{x}$ is the vector of state variables, $\bar{i}$ is the vector of primary inputs, $\text{init}(\bar{x})$ is a propositional formula describing the initial states, and $T(\bar{x}, \bar{i}, \bar{x}')$ is propositional relation describing the transition relation. Primed state variables $\bar{x}'$ denote the next state. Let $post(\bar{x}, \bar{i})$ denote the next state $\bar{x}'$ such that $T(\bar{x}, \bar{i}, \bar{x}')$ holds, and $post(\bar{x})$ denote the set of next states such that for all $\bar{x}' \in post(\bar{x})$, there exists some input $\bar{i}$, such that $T(\bar{x}, \bar{i}, \bar{x}')$ holds. Let $S_{\mathcal{F}}$ denote the state-space of $\mathcal{F}$. Let $C_0, C_1, C_2, \ldots, C_n$ denote the maximal strongly connected components (MSCC) of the pending graph of $S_{\mathcal{F}}$. Let us call each C_i a *supernode* of states. It is well known that the transitions among supernodes $\{C_i\}$ form a directed acyclic graph (DAG). Since the states in the goal region have no outgoing transition, each forms a singleton supernode with incoming transitions only (often called a sink supernode) in the supernode DAG. Abusing notation a little, let us denote the union of sink supernodes of goal states by C_0. Henceforth, C_0 serves as the goal region of the pending graph of $S_{\mathcal{F}}$ (see Figure 5.14 for an illustration). For a state $\bar{x}$, we define $rank(\bar{x})$ as the minimum length of a path from $\bar{x}$ to any state in C_0. For any $\bar{x} \in C_0$, $rank(\bar{x}) = 0$. For a state $\bar{x}$ which has no path to C_0, $rank(\bar{x}) = \infty$. For a supernode C_i, we define $\rho_i = rank(C_i) = \max\{rank(\bar{x}) : \bar{x} \in C_i\}$.

5.6.2 State-Space Characterization and Verification Algorithm

Suppose we want to prove the response property ϕ_r on $\mathcal{F}$, and also suppose that the state-space of $\mathcal{F}$ has the following property:

1. The supernodes have unique ranks, i.e. for any two supernodes C_i, C_j, $\rho_i \neq \rho_j$ for $i \neq j$. Therefore, we assume that indices to any two supernodes C_i, C_j are so assigned that $\rho_i > \rho_j$ for $i > j$.

2. For any C_k in the pending region (*i.e.* for $k > 0$),

   a) $\Psi_1 \triangleq \forall \bar{x} \in C_k \cdot post(\bar{x}) \subseteq C_k \cup C_{k-1}$

   b) $\Psi_2 \triangleq \forall \bar{x} \in C_k \cdot \exists \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in C_{k-1}$

In other words, Ψ_1 means that each pending sub-region C_k has next states either in C_k or in C_{k-1}, and Ψ_2 means that each such C_k must have at least one next state in C_{k-1}. Obviously, states in C_k cannot have any next state with rank greater than ρ_k. Ψ_1 captures this criterion too. If a system meets the above criteria, the pending graph of its state-space is *linearly graded*. A schematic view of such a linearly graded pending graph is shown in Figure 5.10.

Given a pending graph, our objective is to verify that it is linearly graded. If we begin with the goal region C_0, we can recursively reverse engineer all C_k, $k > 0$, using the facts that C_k is the maximal subset of $S \setminus \cup_{t=0}^{k-1} C_t$ that satisfies Ψ_1 and Ψ_2. A pseudo-code is presented in Algorithm 1 below:

5.6.2.1 A Simple Generalization

The above algorithm attempts to discover a linearly graded structure of the pending graph. A pending graph can have other structures that are not linearly graded, but still guarantee

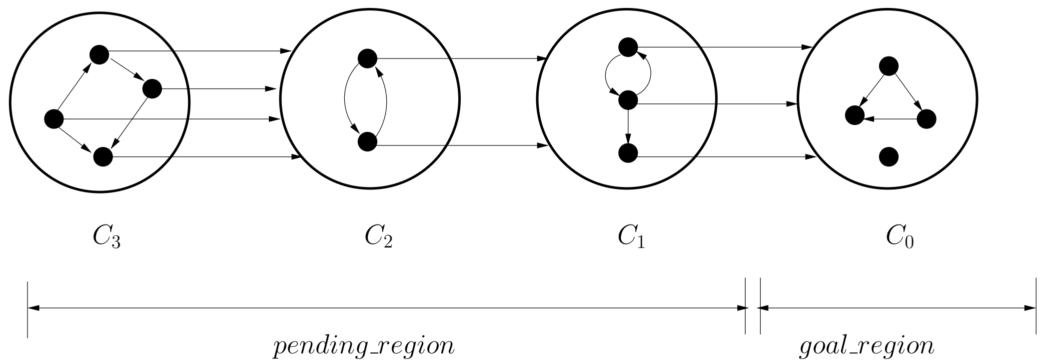


Figure 5.10: Schematic representation of a linearly graded pending graph

---

Algorithm 1: Procedure for finding linear gradation

$C_0 \leftarrow goal\_region$;
$k \leftarrow 0$;
repeat
  $k \leftarrow k + 1$;
  $C_k \leftarrow (\exists \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in C_{k-1}) \cap (\forall \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in C_k \cup C_{k-1})$;
until C_k *is empty*;
if $(S \setminus \cup_{t=0}^{k} C_t) \cap \mathcal{R}$ *is not empty* **then**
  found a *potential* bug (deadlock, or livelock);
else
  any pending state will *eventually* end up in the goal region;
end

---

eventual reachability to the goal region. Figure 5.11 represents an alternative structure which guarantees eventual reachability to the goal region, but is not linearly graded, rather *topologically graded*.

In Figure 5.11, C_4 does not satisfy condition Ψ_1 because $post(C_4) = C_4 \cup C_2 \cup C_1$. Therefore, Algorithm 1 would not cover region C_4, and upon termination, would declare that C_4 can cause a potential liveness bug. However, a simple modification of the criteria Ψ_1 and Ψ_2 can bridge the gap and find that the pending graph of Figure 5.11 also guarantees eventual reachability to the goal region. The new criteria are as follows:

$$\hat{\Psi}_1 \triangleq \forall \bar{x} \in C_k \cdot post(\bar{x}) \subseteq \cup_{\rho_t \leq \rho_k} C_t$$
$$\hat{\Psi}_2 \triangleq \forall \bar{x} \in C_k \cdot \exists \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in \cup_{\rho_t < \rho_k} C_t$$

Algorithm 1 needs only a little modification to use $\hat{\Psi}_1$ and $\hat{\Psi}_2$ in its iteration as presented in Algorithm 2 below.

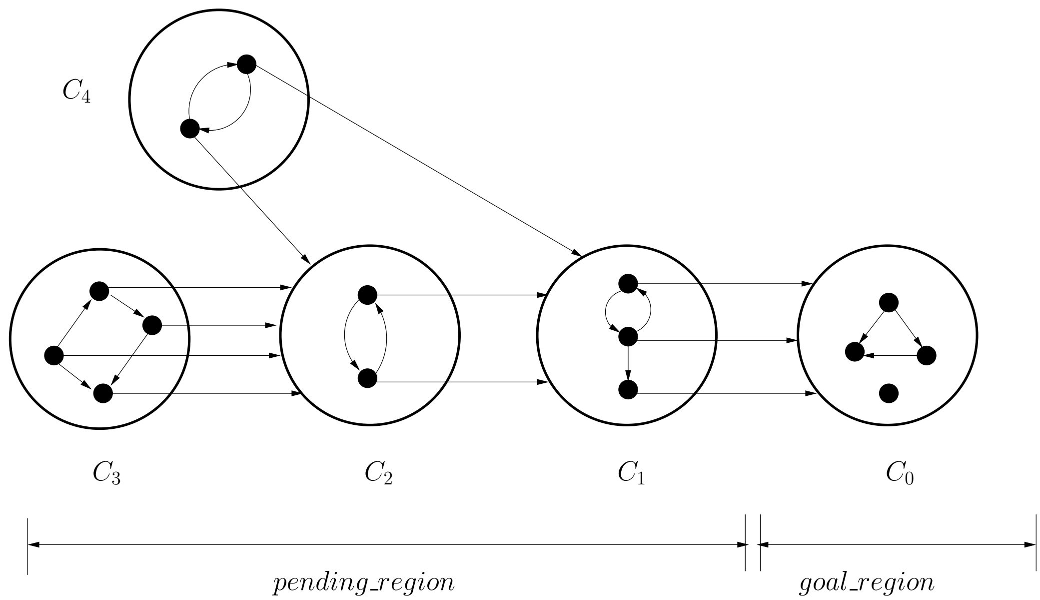


Figure 5.11: Schematic representation of a topologically graded pending graph

Algorithm 2: Procedure for finding topological gradation

$C_0 \leftarrow goalRegion$;
$k \leftarrow 0$;
$G \leftarrow C_0$;
repeat
  $k \leftarrow k + 1$;
  $C_k \leftarrow (\exists \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in C_{k-1}) \cap (\forall \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in C_k \cup G)$;
  $G \leftarrow G \cup C_k$;
until C_k *is empty*;
if $(S \setminus \cup_{t=0}^{k} C_t) \cap \mathcal{R}$ *is not empty* **then**
  found a *potential* bug (deadlock, or livelock);
else
  any pending state will *eventually* end up in the goal region;
end

5.6.2.2 Two Levels of Heuristics

So far, we have discussed the characterizations of linearly graded and topologically graded structures of $S_{\mathcal{F}}$, and have provided general algorithms for testing these properties. While the algorithms discussed are linear in the size of $S_{\mathcal{F}}$, they depend on complex implementations of fix-point algorithms. On the other hand, we observed that a careful extension of *skeleton independent proof* heuristics of [Bradley *et al.*, 2011] yields a surprisingly easy and intuitive proof that $S_{\mathcal{F}}$ is linearly graded for an important class of communication fabrics. The benefits of this proof technique are:

- the proof procedure is simple, and very efficient,
- it involves very little implementation overhead,
- the final proof may be interpreted as a human readable certificate.

Heuristic 1, as presented below, is the core idea of the skeleton independent proof that we use on our benchmarks. *Heuristic 2* is a special heuristic that we developed for our benchmarks to improve on *Heuristic 1*.

5.6.2.3 [Heuristic 1] Skeleton Independent Proof

Suppose we have a special Boolean signal b_1 in our design which partitions the pending region of the pending graph as shown in Figure 5.12. In the figure, A denotes the pending region and G denotes the goal region (i.e. supernode C_0). Suppose b_1 partitions A into two non-empty subsets $A \wedge b_1$ and $A \wedge \neg b_1$ such that (without any loss of generality) $A \wedge b_1$ satisfies both conditions Ψ_1, and Ψ_2, and $A \wedge \neg b_1$ satisfies condition Ψ_2. Then $A \wedge b_1$ may be treated as supernode C_1, and $A \wedge \neg b_1$ as supernode C_2. Hence, signal b_1 partitions the pending region completely into supernodes which expose the linearly graded structure of the pending graph. In the figure, the dotted arrow labeled with Ψ_1 indicates that existence of any such back transition is precluded by Ψ_1.

What if $A \wedge \neg b_1$ does not satisfy Ψ_2? It means $A \wedge \neg b_1$ as a whole cannot be the next partition C_2. If we are lucky, and the design has another special Boolean signal b_2 which partitions $A \wedge \neg b_1$ into $A \wedge \neg b_1 \wedge b_2$, and $A \wedge \neg b_1 \wedge \neg b_2$ such that $A \wedge \neg b_1 \wedge b_2$ satisfies $\Psi_1 \wedge \Psi_2$, and $A \wedge \neg b_1 \wedge \neg b_2$ satisfies Ψ_2. Then again we find a complete partition of the pending graph *viz.* $C_0 = G, C_1 = A \wedge b_1, C_2 = A \wedge \neg b_1 \wedge b_2$, and $C_3 = A \wedge \neg b_1 \wedge \neg b_2$. In this way, an efficient algorithm can be devised that scans a set of designated Boolean signals $B = \{b_1, b_2, \ldots, b_m\}$ in the design, and attempts to reverse engineer the linear gradation of the pending graph. The procedure is shown in Algorithm 3. On termination, it either returns a linear gradation of the pending region using some of the Boolean signals in the given set B; or declares that the given set B is not adequate to find a linear gradation even if such a gradation exists. In that case, we have to fall back on Algorithm 1. A similar argument, using

$\hat{\Psi}_1$, and $\hat{\Psi}_2$ can be applied also to discover a topological gradation of the pending graph using designated Boolean signals.

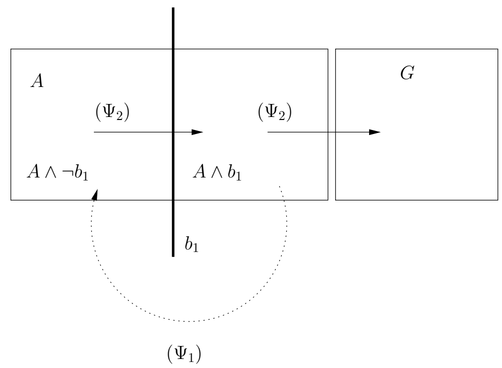


Figure 5.12: Discovering linear gradation using Boolean signals

5.6.2.4 [Heuristic 2] Simplification of first-order formulae

Note that reasoning about Ψ_2, and Ψ_2^* requires either a model checker for first-order logic, or a propositional reasoning engine with efficient quantifier elimination capability. However, based on design insights, we may replace Ψ_2, and Ψ_2^* with simpler propositional formulae Φ_2, and Φ_2^* respectively as shown below.

$$\begin{aligned} \Phi_2 &\triangleq \forall \bar{x} \in (A \wedge \lambda) \cdot post(\bar{x}, 1^{|I|}) \in \neg A \\ \Phi_2^* &\triangleq \forall \bar{x} \in (A \wedge \neg\lambda) \cdot post(\bar{x}, 1^{|I|}) \in \neg A \vee (A \wedge \lambda) \end{aligned}$$

Here $1^{|I|}$ represents the primary input vector where all input signals are assigned to 1.

5.6.2.5 Description of the Algorithm

Algorithm 3 scans a given list of Boolean signals $\mathcal{B}$, and creates an ordered sublist that exposes the linear gradation of the pending graph. It first eliminates (line 3-5) the signals which do not divide the pending region A into two non-empty sub-regions. Inside the while loop (line 6-21), it searches for a literal λ of some signal in the list $\mathcal{B}$ that can serve as an *inductive barrier* (line 8-11). If such literal λ is found, then it checks if the side $A \wedge \lambda$ satisfies Ψ_2 (line 12-14), and the side $A \wedge \neg\lambda$ satisfies Ψ_2^* (line 15-21). If the test in line 12-14 fails, it means λ is not a proper candidate for the barrier, and hence we look for another literal of the same, or some other Boolean signal. On the other hand, if the test in line 15-21 fails, it means that λ is possibly an intermediate inductive barrier of a growing partition. As the next step, we need to try to divide $A \wedge \neg\lambda$ further using other Boolean signal(s) to complete this growing partition. At this stage, we shrink the current list of signals (line 19) and shrink the current pending region (line 20) before looking for other literals.

Algorithm 3: Partitioning using special Boolean Signals

input : Goal Predicate σ_g; special Boolean signals $\{b_1, \ldots, b_n\}$

$A \leftarrow \neg\sigma_g$;
$\mathcal{B} \leftarrow \{b_1, b_2, \ldots, b_n\}$;
forall the $b_i \in \mathcal{B}$ **do**
    if $A \wedge b_i$ *or* $A \wedge \neg b_i$ *is empty* **then**
        $\mathcal{B} \leftarrow \mathcal{B} \backslash \{b_i\}$;
$barrierList \leftarrow \emptyset$;
while $\mathcal{B}$ *is not empty* **do**
    $progress \leftarrow 0$;
    forall the $b_i \in \mathcal{B}$ **do**
        for $\lambda \in \{b_i, \neg b_i\}$ **do**
            $\Psi_1 \leftarrow A \cdot \mathbf{X}(A) \wedge \lambda \Rightarrow \mathbf{X}(\lambda)$;
            if Ψ_1 *is not valid* **then**
                continue; //not a barrier
            $\Psi_2 \leftarrow \forall \bar{x} \in (A \wedge \lambda) \cdot \exists \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in \neg A$;
            if Ψ_2 *is not valid* **then**
                continue; //cannot be the next barrier
            $\Psi_2^* \leftarrow \forall \bar{x} \in (A \wedge \neg\lambda) \cdot \exists \bar{i} \in I \cdot post(\bar{x}, \bar{i}) \in \neg A \vee (A \wedge \lambda)$;
            if Ψ_2^* *is valid* **then**
                report $barrierList$ as the complete partition, and terminate the algorithm;
            else
                append λ to $barrierList$;
                $progress \leftarrow 1$;
                $\mathcal{B} \leftarrow \mathcal{B} \backslash \{b_i\}$;
                $A \leftarrow A \wedge \neg\lambda$;
                continue;
    if $progress = 0$ **then**
        report that set $\mathcal{B}$ is inadequate, and complete partitioning is not possible, hence terminate the algorithm;

Theorem 6. If $\mathcal{B}$ has n Boolean signals, Algorithm 3 makes at most $3n(n-1)$ satisfiability (or validity) checks.

5.6.2.6 Soundness and Completeness

It is easy to see that the algorithm is sound; if it exits from line 19, then the pending graph is indeed linearly graded, and the list of Boolean signals *barrierList* contains a certificate of proof that the state space indeed satisfies property ϕ_r; if the algorithm exits from line 27, it means that the given set of Boolean signals is not informative enough to conclude about the linear gradation of the pending graph, and hence no conclusion about ϕ_r can be drawn. The heuristics of using special Boolean signals to partition the pending region, therefore, makes Algorithm 3 an *incomplete* special case of Algorithm 1 in a sense that Algorithm 3 may not be able to prove that a pending graph is linearly graded even when it is (which could have been established by Algorithm 1). Replacement of $\Psi_2^{(*)}$ with $\Phi_2^{(*)}$ renders it even more incomplete in general. But with these two soundness-preserving heuristics, Algorithm 3 becomes a simple, easy-to-implement procedure that needs only an off-the-shelf propositional SAT solver to establish a proof of liveness. If the underlying state-space indeed admits these special characteristics, we can get a quick and light-weight certificate of proof.

5.6.3 An Illustrative Example : Virtual Channel

We consider $\mathcal{VC}$ of Figure 3.5 to illustrate the notion of linearly graded pending graph for communication fabrics. Any state in the pending graph of $\mathcal{VC}$ will satisfy one of the following mutually disjoint propositional formulae

$$\begin{aligned}\sigma_1 &= \neg Empty(B_1) \wedge \neg Full(B_3)\\ \sigma_2 &= Empty(B_1) \wedge \neg Full(B_3)\\ \sigma_3 &= Empty(B_1) \wedge Full(B_3)\end{aligned}$$

σ_1, σ_2, and σ_3 are such that when $a_1.req$ arrives a grant cannot be produced immediately. $Empty(B)$ and $Full(B)$ are simple predicates defined on buffer B with the following obvious semantics: $Empty(B) = 1$ iff buffer B contains no flit, and $Full(B) = 1$ iff buffer B has no slot free for any more flit. Note that any state satisfying $\neg Empty(B_1) \wedge Full(B_3)$ is unreachable in $\mathcal{VC}$ (see [Chatterjee and Kishinevsky, 2010a] for details), hence not considered.

When $a_1.req$ arrives to $\mathcal{VC}$, it cannot produce $a_1.gnt$ in the same cycle if it is in σ_2 or σ_3; it may produce $a_1.gnt$ in that same cycle if it is in σ_1 depending on the state of the arbiter. When $\mathcal{VC}$ is in σ_3 and the fairness signals $j_1.gnt$ and $m_1.gnt$ are asserted together, the model moves to σ_2 in the next cycle. Otherwise, $\mathcal{VC}$ stays in σ_3. Note that from σ_2, $\mathcal{VC}$ cannot return to σ_3 in the next cycle. Similarly, when $\mathcal{VC}$ is in σ_2 and the fairness signal $i_1.req$ is asserted, it moves to σ_1 in the next cycle, otherwise it stays in σ_2. This behavior is depicted in Figure 5.13. Since σ_1 is the goal region, Figure 5.13 certifies that all states in the pending region will eventually reach the goal region. Hence, ϕ_r holds on $\mathcal{VC}$.

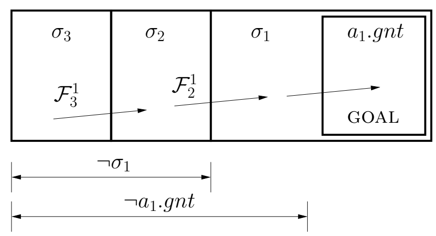


Figure 5.13: Abstract state-space for virtual channel

5.6.3.1 Proof using Algorithm 3

Figure 5.13 shows that the pending graph of the virtual channel is linearly graded. We may use Algorithm 1 to discover σ_2, and σ_3 starting with σ_1. In this section, we will demonstrate the model can be instrumented with some additional Boolean signals, and leverage them as barriers to derive a proof using Algorithm 3. Suppose we introduce six two-bit binary counters $\{Z_i : i = 1, \ldots, 6\}$ in the virtual channel such that the value of Z_i denotes the number of flits in buffer B_i at each time step, for all $i = 1, \ldots, 6$. Let us denote the MSB and the LSB of counter Z_i with Boolean variables z_i^1, and z_i^0 respectively. Since each of the buffers can contain at most 2 flits, each counter Z_i can assume values $z_i^1 z_i^0 = 00$, $z_i^1 z_i^0 = 01$, and $z_i^1 z_i^0 = 10$ only. In terms of the counters Z_i's, we can characterize σ_1, σ_2, and σ_3 as follows (Z_i's are in decimal notation):

$$\begin{aligned}
\sigma_1 &= (Z_1 = 1, Z_3 = 1) \vee (Z_1 = 1, Z_3 = 0) \vee (Z_1 = 2, Z_3 = 0) \\
\sigma_2 &= (Z_1 = 0, Z_3 = 0) \vee (Z_1 = 0, Z_3 = 1) \\
\sigma_3 &= (Z_1 = 0, Z_3 = 2)
\end{aligned}$$

This new abstract state-space is shown in Figure 5.14. In terms of Boolean signals z_i^j, it is easy to see the pending region $A = \sigma_2 \cup \sigma_3$ is partitioned by z_3^1 into two non-empty regions $C_1 = A \wedge \neg z_3^1$ and $C_2 = A \wedge z_3^1$. Note that C_1, and C_2 are two alternative, automatically derived representations of σ_2, and σ_3 respectively. All states in C_1 transit to C_0 once input $i_1.req$ is asserted, and C_2 cannot be reached from C_1 in one step (i.e. C_1 satisfies both Ψ_1, and Ψ_2). On the other hand, all states in C_2 transit to C_1 once inputs $j_1.gnt$ and $m_1.gnt$ are asserted together (i.e. C_2 satisfies Ψ_2). $C_0 \cup C_1 \cup C_2$ covers the whole reachable state-space. Hence, Boolean signal z_3^1 provides a basis for efficient proof of ϕ_r on $\mathcal{VC}$. Experimental results are provided in the following section.

5.6.4 Experimental Results

We ran two separate sets of experiments: in one, the proof of the native liveness property ϕ_r is attempted without using the safety invariants. In the other experiment, the proof of the safety properties in $\mathbb{S}$ are attempted, which aim to replace ϕ_r. For the first set, we used liveness-to-safety transformation to convert the liveness verification problem to an equi-satisfiable

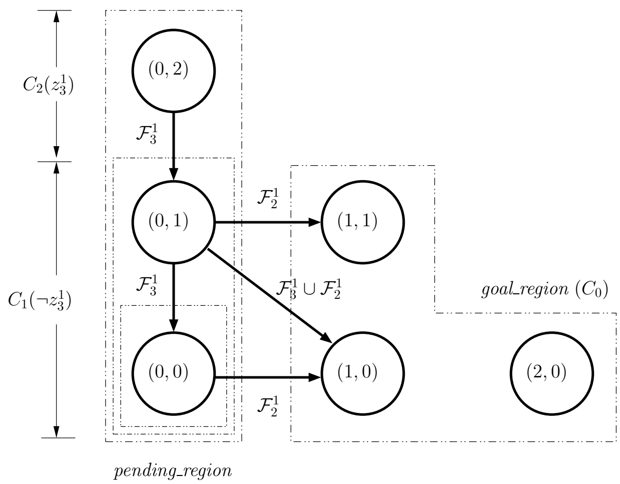


Figure 5.14: Abstract state-space (some transitions pruned) for virtual channel

safety verification problem. In both experiments, we use property directed reachability (PDR) as the safety proof engine on the final single-output safety property instance. Verification run-times of these experiments are shown in Table 5.5. A '-' symbol in place of run-time means ABC could not prove the property within its default resource limits. In Table 5.5, we present run-times for the basic virtual channel, virtual channel with channel buffer, virtual channel with ordering, cascaded virtual channel, and master/slave design.

Design	#PI	#flop	after L2S	$\{\Psi_i\}$
Virtual Channel	8	59	12.78	0.19
Virtual Channel with Buffer	8	77	1515.20	3.01
Virtual Channel with Order	9	104	126.44	4.58
Cascaded Virtual Channel with Buffer(*)	12	182	$>$ 1800	5.72
Simple Master/Slave Design	14	178	-	19.19

Table 5.5: Verification run-times for skeleton independent proof

5.6.5 Conclusion and Future Work

We presented a way of verifying a response property of communication fabrics by using hints from the design. Our proof technique is efficient and scalable. It computes implicit ranking information about the design, thereby revealing interesting insights about the design. We believe this technique exposes interesting avenues of further research.

5.7 Approach IV : Proof based on k-LIVENESS

5.7.1 k-LIVENESS Algorithm

k-LIVENESS is a relatively new algorithm for liveness verification, proposed by Koen Claessen and Niklas Sörensson in [Claessen and Sörensson, 2012]. It has out-performed other state-of-the-art liveness algorithms in the most recent hardware model checking competition [HWM, 2012]. k-LIVENESS is designed to prove an LTL property of the form **FG**p on a finite state system represented as a sequential circuit. Since model checking problem of any ω-regular property can be transformed into an equi-satisfiable model checking problem of another ω-regular property of the form **FG**p using a generalized Büchi automata construction [Wolper *et al.*, 1983], in principle k-LIVENESS serves as a proof technique for the whole family of ω-regular properties. The algorithm works in two phases, a *pre-processing phase* and an *iterative proof phase*. The iterative proof phase is the core proof engine of the algorithm. It is both sound and complete (modulo absence of counterexample) even without the pre-processing phase. However, in some cases the pre-processing phase significantly simplifies and expedites the iterative proof phase. Below, we describe the iterative proof phase first, then describe the pre-processing phase.

5.7.1.1 Iterative Proof Phase

Intuition 1. Suppose property **FG**p holds on a design for some design signal p. It means that signal p may toggle initially, but after some (finite) time, it will assume logic value 1 and remain 1 forever. Figure 5.15 depicts the waveform of a typical signal p which satisfies **FG**p. In other words, if **FG**p holds on the design, p assumes logic value 0 for at most a finite

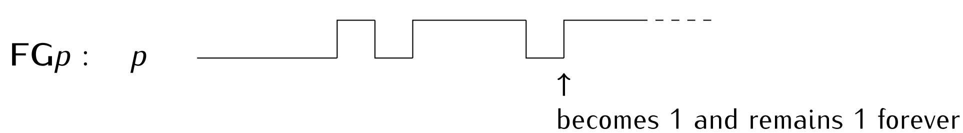


Figure 5.15: A typical waveform for a signal p that satisfies property FGp

number k_{max} times. In the iterative proof phase, k-LIVENESS searches for this k_{max} iterating over non-negative integers starting from 0. At the i-th iteration, for $i \geq 0$, it verifies if signal p can assume logic value 0 for no more than i times. It is a safety verification obligation. If a safety verification tool proves this obligation, then $k_{max} = i$ and **FG**p is proved on the design. Otherwise k-LIVENESS moves to the next iteration and checks if p can assume logic value 0 for no more than $i + 1$ times.

Proof Iterations. The above intuition is realized by k-LIVENESS with the help of an auxiliary circuit called an *absorber circuit* as shown in Figure 5.16. Suppose we want to prove **FG**p

on a design. In the first iteration, k-LIVENESS attempts to prove (on the original design) that signal p can assume logic value 0 no more than 0 times, *i.e.* p is a 'constant 1' signal. Any safety engine can be discharged to prove this property. If the proof goes through, k-LIVENESS terminates declaring that **FG**p holds on the design. If the proof fails, then either **FG**p does not hold on the design, or there exists at least one computation where p assumes logic value 0 for one or more times. k-LIVENESS thus moves to the second iteration and attempts to prove that p can assume logic value 0 for no more than once. For this, it attaches the absorber circuit of Figure 5.16 at signal p on top of the original design. The absorber circuit, whose functionality is described within the text-box on the next page, is designed so that the same signal p is propagated to p_{out} except for the first occurrence of logic value 0 on p (see Figure 5.17 for an example). Hence, if p assumes no more than one logic value 0, p_{out} would behave as a 'constant 1' signal. If k-LIVENESS can prove this claim with the help of a safety verification engine, then again it proves that **FG**p holds on the design. Otherwise, no decision can be made and the process moves to the next iteration adding another absorber circuit to the output.

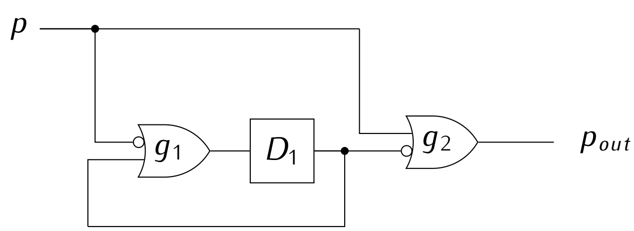


Figure 5.16: Absorber Logic that absorbs one 'drop' on p

Working Principle of Absorber Circuit

Figure 5.16 shows one copy of absorber circuit which 'absorbs' the first logic value 0 from the input signal p. The register D_1 is initialized to 0. If p becomes 0 for the first time at time step t (*i.e.* the first 'drop' from logic value 1 in p's waveform, see Figure 5.17), D_1 gets set to 1 at time $t+1$ and remains 1 forever. Note that D_1 controls the output of OR gate g_2 until time t and forces p_{out} to remain 1 so far. This value matches with p up to time $t-1$ as p becomes 0 for the first time at time t. At t, p drops to 0, but p_{out} remains 1 due to $D_1 = 0$. This is how the circuit absorbs the first 0 from p. Form $t+1$ onwards, D_1 loses control over g_2 and p propagates to p_{out} as it is. Clearly, if p has $k \geq 1$ drops in its waveform, the absorber circuit absorbs the first one and results in $k-1$ drops in the waveform of p_{out}. As a corollary, if p has at most one drop in its waveform, p_{out} becomes a 'constant 1' signal.

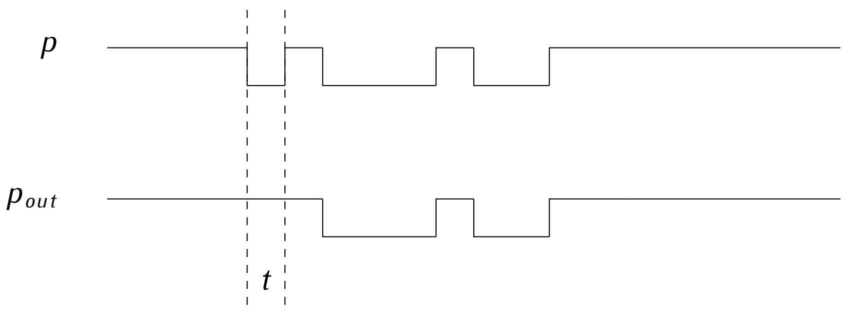


Figure 5.17: A typical waveform for signal p that enters the absorber logic of Figure 5.16 and the corresponding waveform for p_{out}

In the third iteration, k-LIVENESS attempts to prove that p can assume logic value 0 for no more than twice. For this, it attaches two copies of absorber circuit at signal p of the original design. The resulting cascade of absorber circuits is shown in Figure 5.18. The output signal p_{out} is now a 'constant 1' signal if and only if p assumes logic value 0 for no more than twice. If a safety verification engine can prove the claim, we are done. Otherwise, another absorber logic is added at the end of the cascade of Figure 5.18 and the process continues.

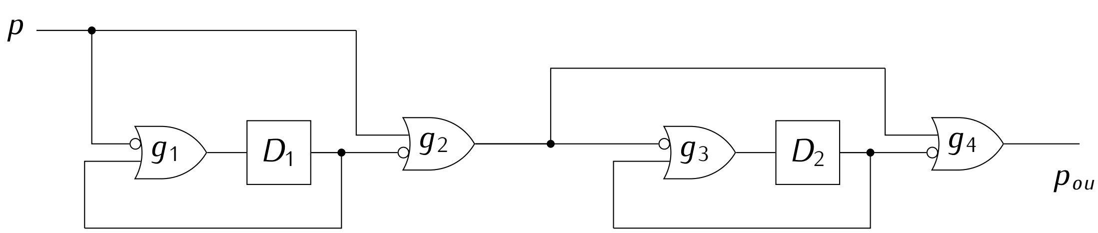


Figure 5.18: Cascade of absorber logic for absorbing 2 'drops' on p

For example, let signal p from Figure 5.17 be fed to the two-level cascade of absorber circuits as shown in Figure 5.18, the resulting waveform of p_{out} takes the shape shown in Figure 5.19. Here p drops to 0 for the first time at $t \geq 0$ and for the second time at $t' > t$. Note in Figure 5.19 that p drops to 0 more than twice. Hence, p_{out} is not a 'constant 1' signal in this case.

General Case and Soundness: For the i-th iteration ($i \geq 2$), k-LIVENESS adds a cascade of $i - 1$ copies of the absorber logic as in Figure 5.18. If a safety verification engine proves that p_{out} of the resulting circuit is 'constant 1', then we know that p cannot assume logic value 0 for more than $i - 1$ times. In other words, p will eventually be stuck at logic value 1. k-LIVENESS can, therefore, declare that **FG**p holds on the design and terminate. This justifies soundness of k-LIVENESS algorithm.

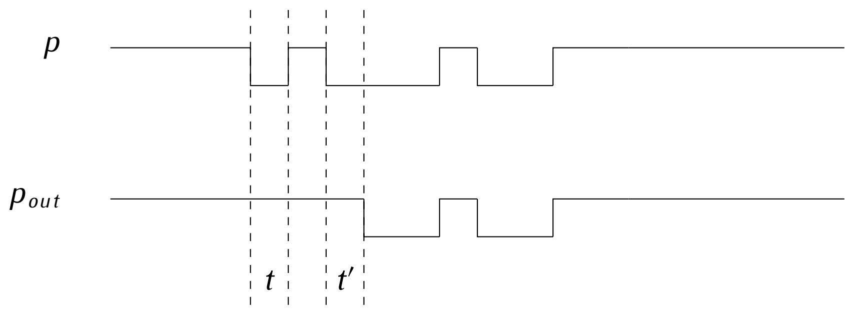


Figure 5.19: Waveform for p and p_{out} obtained from a two-level cascade of absorber circuit

5.7.1.2 Pre-processing Phase

Intuition 2. For a finite state system, an infinite computation is always resulted by a lasso loop in the state transition graph. For $\mathbf{FG}p$ to hold on a finite state system, the latter cannot contain a lasso loop starting from an initial state whose loop part has a state with $p = 0$. We call such a lasso loop a 'bad loop' and we want to prove that such a bad loop does not exist in the state space. Suppose there is another signal s in the design such that $\mathbf{FG}s$ is *known* to hold. That means s will eventually get stuck at 1 along any computation. We claim that *for any lasso loop starting from an initial state, its loop part cannot contain a state with* $s = 0$. This is because a loop state with $s = 0$ means that it is possible to come back to that state again and again, but since s will eventually get stuck at 1 along the computation, visiting back a state with $s = 0$ is impossible after some finite time steps. Signal s, therefore, acts as a constraint such that any reachable loop must remain confined in the $s = 1$ part of the state space. This information can be used to constrain the original query if there is a bad loop containing at least one $p = 0$ state. □

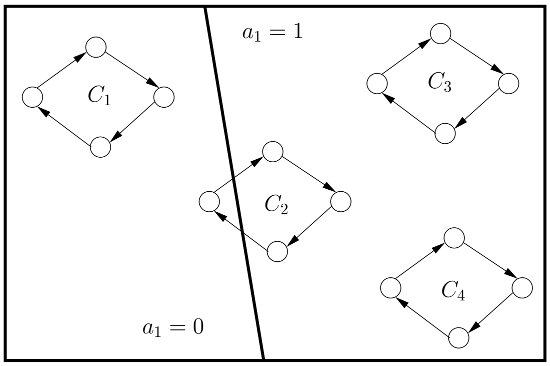
Intuition 2 is the key for the pre-processing phase of the k-LIVENESS algorithm. It calls constraints of the form $\mathbf{FG}s$ *stabilizing constraints* and mines them automatically from a design. For computational advantage, k-LIVENESS restricts its attention only to a syntactic subset of candidate stabilizing constraints. It uses the following rules for identification of stabilizing constraints:

When a is a design signal (i.e. either a gate output, or a register output of AIG S),

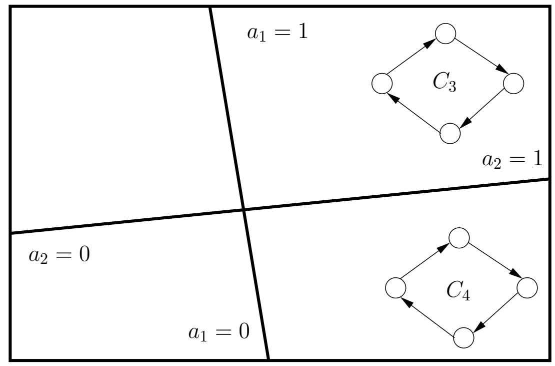
(R1) if $S \models \mathbf{G}(a \Rightarrow \mathbf{X}(a))$, or $S \models \mathbf{FG}(a \Rightarrow \mathbf{X}(a))$, then $\mathbf{FG}(a = \mathbf{X}(a))$ is a stabilizing constraint,

(R2) suppose $S \models \mathbf{FG}(a \Rightarrow \mathbf{X}(a))$ and additionally $S \models \mathbf{FG}(a \Rightarrow p)$ holds where $S \models \mathbf{FG}p$ is our original liveness proof obligation, then $\mathbf{FG}\neg a$ is a stabilizing constraint,

(R3) dual to rule (R2), suppose $S \models \mathbf{FG}(a \Rightarrow \mathbf{X}(a))$ and additionally $S \models \mathbf{FG}(\neg a \Rightarrow p)$, then $\mathbf{FG}a$ is a stabilizing constraint,



(a) Arena created by a_1



(b) Arena created by a_1 and a_2

Figure 5.20: Formation of arena with stabilizing constraints and their role in precluding unreachable loops

Claessen and Sörensson proposed an induction based approach to quickly discover an under-approximation of the set of all stabilizing constraints present in a design. If $\mathcal{W}_s = \{w_1, w_2, \ldots, w_n\}$ is the set of stabilizing constraints identified by some algorithm based on the above three rules, k-LIVENESS is invoked on AIG S to prove the constrained property $\mathbf{FG}(\wedge_{i=1}^{n} w_i \Rightarrow p)$. Referring to the connection between stabilizing properties, arenas and walls, predicate $\wedge_{i=1}^{n} w_i$ characterizes the only interesting stabilizing arena that may contain bad loops. Constraining the original query with this new predicate eliminates unnecessary search in the other stabilizing arenas.

Wall, Arena and Stabilizing Constraints

Let the rectangles of Figure 5.20 represent the state space of a design and C_1, C_2, C_3 and C_4 represent various loops in the underlying state transition graph. Now, suppose the design has a signal a_1 which is stabilizing. Moreover, suppose the predicates $(a_1 == 0)$ and $(a_1 == 1)$ divide the state space in such a way that loop C_1 belongs completely to the partition $(a_1 == 0)$, loop C_2 has states in both the partitions and loop C_3 and C_4 belong completely to the partition $(a_1 == 1)$ (see Figure 5.20(a)). Since a_1 is a stabilizing signal, as per Intuition 2, loops C_1 and C_2 are either infeasible, or unreachable. Now, suppose that the design has another stabilizing signal a_2 such that loop C_4 belongs to the partition $(a_2 == 0)$. By Intuition 2 applied to a_2, C_4 becomes unreachable as well, leaving C_3 as the only possible candidate for reachable loops. The stabilizing nature of a_1 and a_2, therefore focuses the search for bad loops only to the partition $(a_1 == 1 \wedge a_2 == 1)$. Bradley *et al.* coined the terms *wall* and *arena* in [Bradley *et al.*, 2011]. A *wall* is a Boolean predicate which defines a region in the state space such that no loop can cross the region boundary. Clearly, a wall divides the state space into two SCC-closed regions. If $\mathcal{W}$ is a given set of walls, then it divides the state space into $2^{|\mathcal{W}|}$ SCC-closed regions, each of them is called an *arena*. Stabilizing constraints are stronger in notion than walls. If $\mathcal{W}_s$ is a set of stabilizing constraints, we say that $\mathcal{W}_s$ divides the state space into $2^{|\mathcal{W}_s|}$ *stabilizing arenas*. Interestingly, only one among them will contain potentially reachable loops. No reachable loop can exist in the remaining $2^{|\mathcal{W}_s|} - 1$ stabilizing arenas. As in Figure 5.20(b), signals a_1 and a_2 divide the state space into four stabilizing arenas. The only interesting arena that can contain reachable loops is $(a_1 == 1 \wedge a_2 == 1)$. This way, a set of stabilizing constraints can trim away significant parts of the search space, and help k-LIVENESS converge faster.

5.7.2 Disjunctive Stabilizing Constraints

In order to prove response properties of communication fabrics, we extend the notion of stabilizing constraints as used in [Claessen and Sörensson, 2012] by generalizing the domain of its candidates. As proposed in rule (R1) above, k-LIVENESS scans over all signals present in the design and checks which satisfy (R1); it tests all gate outputs and register outputs of the AIG representing the design. We observe that some predicates (*i.e.* Boolean functions) defined over some subset of design signals may also satisfy (R1) and may simplify the subsequent proof obligations significantly. These predicates can range over simple functions like conjunction, implication, or complex, not-so-intuitive predicates. Unfortunately, if these predicates were not present as signals in the AIG, the native pre-processing step of k-LIVENESS as implemented in [Claessen and Sörensson, 2012] would not find them to leverage them in the proof. With this insight, we propose to enrich the domain of candidate signals for the test of rule (R1) such that the tester would also investigate some Boolean combinations of existing signals.

Generally, knowing which combination of design signals to consider for their stabilizing behavior is difficult; we neither know the definitions of the target Boolean functions, nor their supports or support sizes. However, in the application domain of response verification of communication fabrics, we can hypothesize a particular class of Boolean function that can exhibit stabilizing behavior. This class has the following characterization: suppose a_1 is a design signal that satisfies $S \models \mathbf{G}(a_1 \Rightarrow \mathbf{X}(a_1))$. For such an a_1, we would like to find another signal a_2 such that $S \not\models \mathbf{G}(a_2 \Rightarrow \mathbf{X}(a_2))$, but $S \models \mathbf{G}((a_1 \vee a_2) \Rightarrow \mathbf{X}(a_1 \vee a_2))$. We call signals that can qualify as a_1 '*level-1 stabilizing constraints*' and signals that can qualify as a_2 '*level-2 stabilizing constraints w.r.t. a_1*'. In general, a *level-τ stabilizing constraint* can be defined. It is defined recursively as follows:

1. A signal a is a level-1 stabilizing constraint iff $S \models \mathbf{G}(a_1 \Rightarrow \mathbf{X}(a_1))$
2. A signal a_τ is a level-τ stabilizing constraint, for $\tau > 1$, iff $S \models \mathbf{G}((a_1 \vee a_2 \vee \ldots \vee a_\tau) \Rightarrow \mathbf{X}(a_1 \vee a_2 \vee \ldots \vee a_\tau))$, but a_τ is not a level-$(\tau - 1)$ stabilizing constraint.

In the original k-LIVENESS algorithm, only level-1 stabilizing constraints were considered in the pre-processing step. In our pre-processor, our objective is to consider up to level-τ for some $\tau > 1$. We set τ as a parameter whose value would depend on the available computational resource. Algorithm 4 presents a procedure for finding level-τ or less stabilizing constraints.

Algorithm 4: Identification of stabilizing constraints

C : set of candidate signals;
$L_0 \leftarrow \emptyset$;
$\tau \leftarrow 1$;
repeat
    $\tau \leftarrow \tau + 1$;
    $M_\tau \leftarrow \cup_{i=0}^{\tau-1}\{x|x$ appears in any one of the disjuncts in $L_i\}$;
    $C \leftarrow C \backslash M_\tau$;
    forall the $a \in C$ **do**
        forall the $l \in L_{\tau-1}$ **do**
            if $S \models \boldsymbol{G}((l \vee a) \Rightarrow \boldsymbol{X}(l \vee a))$ **then**
                $L_\tau \leftarrow L_\tau \cup \langle l, a\rangle$;
until $(\tau >$ THRESHOLD$) \vee (C = \emptyset) \vee (L_\tau = \emptyset)$;

Note that for a level-τ stabilizing constraint a_τ, signal $a(\tau) = a_1 \vee a_2 \vee \ldots \vee a_\tau$ becomes a level-1 stabilizing constraint, with a difference that $a(\tau)$ may not be present in the original

AIG, though signals a_1 through a_τ are present. Therefore, our augmented domain of candidate signals is the disjunctive domain over existing signals with parameterized support size. We specialize this disjunctive domain further by imposing the (recursive) restriction that a_τ cannot be a level-($\tau - 1$) stabilizing constraint. This restriction stems from an observation that in communication fabrics, for a level-1 stabilizing constraint a_1, there might exists a (level-2 stabilizing) signal a_2, such that $\neg a_1 \Rightarrow a_2$ is a level-1 stabilizing signal. Similarly, there might exist a (level-3 stabilizing) signal a_3 such that $\neg a_1 \Rightarrow (\neg a_2 \Rightarrow a_3)$ is again a level-1 stabilizing signal. Note that the general form $\neg a_1 \Rightarrow (\neg a_2 \Rightarrow \ldots (\neg a_{\tau-1} \Rightarrow a_\tau)) \ldots)$ is equivalent to $a_1 \vee a_2 \vee \ldots \vee a_\tau$. Apart from capturing this communication fabric specific requirement, this restriction over the general disjunctive domain prevents a potential blow-up associated with considering all possible disjunctions for some support size.

5.7.3 xMAS-**specific predicates**

For xMAS communication fabrics, we observe that the information whether some FIFO buffer is empty or not, or whether some buffer is full or not, has a very natural connection to how the fabric eventually issues a grant to a request. However, in a bit-blasted fabric this information is not readily available. A specialized algorithm is required that would identify appropriate register variables (associated with a particular FIFO buffer) and then synthesize appropriate Boolean functions corresponding to a buffer being empty or not etc. Without any prior hint, it would be challenging for a general purpose liveness algorithm, like k-LIVENESS, to find such specialized information efficiently. Based on this observation, we propose to introduce this information as explicit predicate signals in the design. In particular, for each buffer B in the fabric, we introduce predicates $is\_empty(B)$ and $is\_full(B)$ in the AIG. The predicates have the following obvious definitions: $is\_empty(B) = 1$ iff buffer B is empty; $is\_full(B) = 1$ iff buffer B is full. During our experiments, before we invoke Algorithm 4 to discover stabilizing constraints for a design, we initialize the set of candidate signals C as $\{is\_empty(B), \neg is\_empty(B), is\_full(B), \neg is\_full(B) | B$ is a buffer in the fabric$\}$.

5.7.4 k-LIVENESS **algorithm for response verification**

As an ω-regular property, model checking problem for ϕ_r can be transformed into another model checking problem of a property of the form of **FG**p using a generalized Büchi automata construction. However, we can use the core idea of k-LIVENESS to prove ϕ_r in a more direct way that avoids such a translation. We illustrate this using the example of $\mathcal{VC}$ as follows:

If ϕ_r is to hold on $\mathcal{VC}$, *i.e.* if every $a_1.req$ is eventually followed by a $a_1.gnt$, the interval for which $\mathcal{VC}$ waits for $a_1.gnt$ once $a_1.req$ is asserted (called the *pending interval*) can never be infinite. Thus, the number of times the fairness signals are asserted during any pending interval is finite also. So, k-LIVENESS algorithm can be used to prove that the number of times the fairness signals are asserted during any pending interval is finite for $\mathcal{VC}$.

A monitor that implements the above idea can be constructed as shown in Algorithm 5. This monitor is attached to $\mathcal{VC}$ and k-LIVENESS is called to prove that signal *allFairCount* can assume logic value 0 only finitely many times. Algorithm 5 is written in an imperative pseudocode with standard semantics where all assignments are to be evaluated in parallel at every clock. We adopt Verilog-style semantics for the data-types, namely, **wire** represents combinational signals and **reg** represents sequential signals (*i.e.* registers). All **reg** variables are initialized to logic value 0 and for a **reg** variable r, **next**(r) computes its next state value.

Algorithm 5: k-LIVENESS Monitor for proving ϕ_r on $\mathcal{VC}$

```
wire pending, pendingInterval, allFair;
wire fair[6] = {i1.req, i2.req, j1.gnt, j2.gnt, m1.gnt, m2.gnt};
reg oracleSaved, gntSaved, fairFlop[6];
pending := a1.req ∧ ¬a1.gnt;
next(oracleSaved) := oracleSaved ∨ (oracle ∧ ¬oracleSaved ∧ pending);
next(gntSaved) := gntSaved ∨ (oracleSaved ∧ a1.gnt);
pendingInterval := oracleSaved ∧ ¬gntSaved;
allFair := ⋀_{i=1}^{6} fairFlop[i];
for i ∈ {1, 2, ..., 6} do
    if allFair then
        next(fairFlop[i]) := 0;
    else
        next(fairFlop[i]) := fairFlop[i] ∨ fair[i];
allFairCount := ¬(pendingInterval ∧ allFair);
```

Algorithm 5 demonstrates how the core idea of k-LIVENESS can be used to verify ϕ_r in a direct way, without constructing a generalized Büchi automata for the whole property ϕ_r. However, this basic monitor of Algorithm 5 does not leverage the stabilizing constraints proposed in Section 5.7.2. Below, we show how Algorithm 6 modifies Algorithm 5 to include the stabilizing constraints. In Algorithm 6, M stands for the set of all stabilizing constraints mined in the pre-processing step. The roles of stabilizing constraints is captured in the variable *arenaViolation*.

5.7.4.1 Working principle and correctness of the algorithms

In both Algorithm 5 and Algorithm 6, we use a primary input *oracle* to model the non-deterministic choice for an arbitrary pending interval. It triggers the monitor at an arbitrary time step when $a_1.req$ is asserted, but $a_1.gnt$ is not. Once this event occurs, register *oracle-Saved* remembers the event and disables *oracle* forever. This event also opens the pending

Algorithm 6: Arena-aware k-LIVENESS monitor

wire *pending, pendingInterval, allFair*;
wire $fair[6] = \{i_1.req, i_2.req, j_1.gnt, j_2.gnt, m_1.gnt, m_2.gnt\}$;
reg $f_1, f_2, \ldots, f_{|M|}$;
reg *oracleSaved, gntSaved, fairFlop*[6];
forall the $m \in M$ **do**
  if $oracle \wedge \neg oracleSaved \wedge pending$ **then**
    next$(f_m) := m$;
  else
    next$(f_m) := f_m$;

$arenaViolation := \bigvee_{m \in M} (f_m \neq m)$;
$pending := a_1.req \wedge \neg a_1.gnt$;
next$(oracleSaved) := oracleSaved \vee (oracle \wedge \neg oracleSaved \wedge pending)$;
next$(gntSaved) := gntSaved \vee (oracleSaved \wedge (a_1.gnt \vee arenaViolation))$;
$pendingInterval := oracleSaved \wedge \neg gntSaved$;
$allFair := \bigwedge_{i=1}^{6} fairFlop[i]$;
for $i \in \{1, 2, \ldots, 6\}$ **do**
  if $allFair$ **then**
    next$(fairFlop[i]) := 0$;
  else
    next$(fairFlop[i]) := fairFlop[i] \vee fair[i]$;

$allFairCount := \neg(pendingInterval \wedge allFair)$;

interval under examination. This interval is closed in Algorithm 5 when a subsequent $a_1.gnt$ arrives. Note that this construction works because there is no obligation of matching an $a_1.gnt$ with a corresponding $a_1.req$ and $a_1.req$ has the persistence property [Chatterjee and Kishinevsky, 2010a]. Algorithm 6 differs from Algorithm 5 in the way it closes the pending interval. When it opens the pending interval, it takes a snap-shot of all signals of the form $a_1 \vee \ldots \vee a_\tau$ in registers $\{f_m\}$ where signals a_i are level-i stabilizing constraints discovered in the pre-processing phase. Then through variable *arenaViolation*, it keeps track of whether any one such signal has changed its value (*i.e.* the check $f_m \neq m$). Since each such signal cannot flip its value within a potential counter-example to response property, a scenario under which it changes its value cannot be a candidate for violation to ϕ_r. If any one such signal has changed its value, then the algorithm closes the pending interval (through $a_1.gnt \vee arenaViolation$). Note that Algorithm 6 achieves an early closure of the pending interval compared to Algorithm 5 due to the use of the disjunctive stabilizing constraints. This helps to reduce the number of iterations in the proof phase. It is straight-forward to see that these constructions work for the general situation of response verification as well.

5.7.5 Experimental Results

We use ABC [Mishchenko, 2013] as our environment for both safety and liveness verification. We developed a Verilog implementation of xMAS library closely following the logical definitions given in [Chatterjee *et al.*, 2010], and used this library to implement various models of communication fabrics. We use an in-house tool VERIABC [Long *et al.*, 2011] to bit-blast these Verilog models. VERIABC uses Verific, a commercial Verilog front-end analyzer, to read in the Verilog models. The benchmarks considered are industrially relevant, yet their designs are available in the literature ([Chatterjee and Kishinevsky, 2010a], [Gotmanov *et al.*, 2011], [Chatterjee *et al.*, 2010]). They represent the basic virtual channel (VC), the virtual channel with channel buffer (VCB), the virtual channel with ordering (VCO) and the cascaded virtual channel with ordering (CVCO). CVCO is a cascade of two $\mathcal{VCO}$ connected in series.

In [Claessen and Sörensson, 2012], the authors compared their technique with liveness-to-safety conversion [Schuppan and Biere, 2004]. They demonstrated that the latter is a close contender of the k-LIVENESS algorithm. The liveness-to-safety transformation converts the liveness verification problem to an equi-satisfiable safety verification problem. It is a promising liveness verification technique currently adopted in industry to solve challenging verification problems of industrial importance [Baumgartner and Mony, 2009]. For comparison, we ran our benchmarks with our own implementation of liveness-to-safety; results are shown in Table 5.6. We used property directed reachability (PDR, a.k.a. IC3) [Bradley, 2011] as the safety proof-engine on the final single-output safety property instance. Verification run-times are shown in seconds in column 'after L2S'. We used a time-out of 1800 seconds. The times taken by liveness-to-safety engine to solve these benchmarks, in spite of their relatively small register counts, demonstrate the inherent complexity of the liveness problems being addressed.

Design	#PI	#flop	after L2S
Virtual Channel (VC)	8	59	12.78
Virtual Channel with Buffer (VCB)	8	77	1515.20
Virtual Channel with Order (VCO)	9	104	126.44
Cascaded Virtual Channel with Order (CVCO)	12	182	> 1800

Table 5.6: Liveness property verification run-time with liveness-to-safety transformation

Performance of our extended k-LIVENESS algorithm on these benchmarks is presented in Tables 5.7 and 5.8. Table 5.7 shows the experiments that use arenas, while Table 5.8 shows the experiments that do not use arena violations. PI and #flop columns in both tables represent the number of primary inputs and number of registers in the designs. Note that these numbers are slightly different in Tables 5.6, 5.7 and 5.8 for the same design. This is due to property specific logic simplification performed by veriABC. Columns PPT, PT, k and #c of Table 5.7 represent respectively pre-processing times (*i.e.* time to generate stabilizing constraints), proof time (*i.e.* time required to prove the final property), iterations performed by k-liveness in order to get the proof and the total number of stabilizing constraints generated in the pre-processing phase. Times in columns PPT and PT are in seconds. Similarly, columns PT, k' and #c of Table 5.8 represent respectively time required to get the final proof, number of iterations performed by k-liveness and the number of stabilizing constraints generated. Note that #c columns of Table 5.7 and Table 5.8 are identical because the same set of stabilizing constraints were used in both the experiments. For this reason, we have omitted the PPT column from Table 5.8 because the same set of stabilizing constraints as in Table 5.7 were re-used in Table 5.8 without re-generating them. The benefit of using disjunctive stabilizing constraints is reflected in column k and k' of these two tables. Note that $k < k'$ for every designs considered, which shows the success of disjunctive monotone constraints in reducing k, sometimes quite remarkably. This also reduces run-time (column PT) of the proof step, sometimes quite significantly. As the use of disjunctive monotone constraints achieves the proof with smaller k, it puts less stress on the safety verification engines and increases the chance of convergence of the final proof for challenging benchmarks. For proving all safety obligations, we use the property directed reachability (PDR) engine of ABC. In the pre-processing phase, we discharge a number of verification obligations. We use PDR to prove these one after another sequentially. We used only level-1 and level-2 stabilizing constraints in our experiments.

Design	PI	#flop	PPT	PT	k	#c
VC	13	161	7.32	0.32	1	37
VCB	13	217	267.8	5.31	1	54
VCO	14	210	23.53	12.95	2	47
CVCO	21	386	638.28	230.36	3	82

Table 5.7: Performance of k-liveness with arenaViolation

Design	PI	#flop	PT	k'	#c
VC	13	124	0.55	2	37
VCB	13	163	12.54	2	54
VCO	15	163	15.74	4	47
CVCO	21	303	2711.71	6	82

Table 5.8: Performance of k-liveness without arenaViolation

Note that the pre-processing step for discovery of level-1 and level-2 disjunctive stabilizing constraints took significant time, as shown in column PPT of Table 5.7. We believe that further improvements in implementation of this step can reduce this run-time. We used PDR as the proof tool to check validity of rule (R1) in this phase. A more light-weight tool like induction could be tried to improve this run-time. We used a set of known design invariants [Chatterjee and Kishinevsky, 2010a] as constraints during PDR-based analysis in order to restrict the state-space and find a proof faster. We believe that in some cases, these invariants were very effective in reducing the state-space (for example, the VCO). Whereas in some other case, the supplied invariants were not descriptive enough to help PDR find a proof faster (for example, VCB). A direction for speeding up the pre-processing step might be to supply as many helpful safety invariants as possible. However, we would like to emphasize that in spite of room for further improvement in run-time of this pre-processing phase, our extended k-LIVENESS algorithm managed to finish the whole proof process faster than the cases where we did not use the stabilizing constraints (*i.e.* Table 5.8). For example, the total time of solving benchmark CVCO with level-1 and level-2 stabilizing constraints (PPT+PT of Table 5.7) is less than the proof time alone of Table 5.8.

5.7.6 Related Work

Ranking-oriented proof of liveness is certainly not a new research topic. The underlying core mathematical notions, *viz. well-founded induction* and *ranking function*, are fundamental concepts in discrete mathematics. Based on these ideas, Zohar Manna and Amir Pnueli laid the foundation of ranking-oriented proof of liveness of general reactive systems in the 1980's [Manna and Pnueli, 2010]. Recently, research interests in this area have been renewed, particularly in the context of termination analysis of software ([Cook *et al.*, 2006], [Cook *et al.*, 2007], [Ben-Amram, 2009], [Podelski and Rybalchenko, 2004]). To the best of our knowledge, however, no prior work has addressed the problem of ranking-oriented proof of liveness for communication fabrics.

Among recent work on liveness verification of communication fabrics, most closely related to our problem are [Gotmanov *et al.*, 2011] and [Verbeek and Schmaltz, 2011]. Both of these two are based on xMAS formalism. In [Gotmanov *et al.*, 2011], Gotmanov et. al. proposed an efficient technique for proving deadlock freedom of communication fabrics by composing sets of sufficient conditions for deadlock freedom of each individual xMAS component. This system-wide condition is checked with a SAT solver for unsatisfiability. Our algorithm also addresses the same problem, with a focus on verification of response properties. In [Verbeek and Schmaltz, 2011], Schmaltz et al. proposed algorithms based on graph analysis that detects deadlock freedom for network-on-chips represented using xMAS formalism. Their algorithm does not address the problem of bit-level verification, rather it certifies designs at the micro-architecture level.

Research on general algorithms for bit-level liveness verification has gained remarkable momentum in recent times. In hardware model checking world, new algorithms and tools are being developed. Biere et al. showed how algorithms for safety verification can be re-used for liveness verification through their liveness-to-safety conversion [Biere *et al.*, 2002; ?]. Bradley et. al. proposed FAIR [Bradley *et al.*, 2011], a scalable, incremental algorithm for liveness verification based on their remarkably successful algorithm IC3 [Bradley, 2011] for safety verification. k-LIVENESS is a recent addition to the arsenal of scalable liveness algorithms. Both k-LIVENESS and our methodology have strong conceptual connection with FAIR. The term (and the notion of) 'arena' used in Algorithm 6 is inspired by FAIR.

5.7.7 Conclusion and Future Work

We presented a generalization of the pre-processing phase of the new k-LIVENESS algorithm. k-LIVENESS has established itself as the state-of-the-art of bit-level liveness verification algorithm by out-performing existing algorithms in the liveness track of the recent hardware model checking competition. By generalizing a key step, we improved its effectiveness further on challenging benchmarks. Our generalization, called disjunctive stabilizing constraints, was inspired by applications of liveness verification for communication fabrics. We experimented on fabric designs of industrial relevance and demonstrated effectiveness of our approach. We believe this is only a first step and opens up a new avenue of research. Disjunctive stabilizing constraint is a general notion that could be effective outside communication fabric application. Developing a well-engineered implementation of a mining algorithm for disjunctive constraints is our next step. It would be interesting to investigate which other application domains can benefit from this extended pre-processing step of the k-LIVENESS algorithm. In our implementation, we used PDR to filter out signals that are not stabilizing. An induction based approach could discover an adequate subset of all stabilizing signals up to level-τ rather quickly. It needs further experimentation to decide which technique should be used for challenging benchmarks. As a theoretical question, it would be interesting to investigate the formal connection between level-τ stabilizing constraints and the classical notion of ranking functions. In our extension of the k-LIVENESS algorithm, we only considered rule (R1). It would be interesting to determine the roles of rules (R2) and (R3) in the definition of level-τ stabilizing constraints.

Chapter 6

Structural Invariant Generation

6.1 Introduction

In [Chatterjee and Kishinevsky, 2010a] (and in its extended version [Chatterjee and Kishinevsky, 2012]), Chatterjee and Kishinevsky presented an algorithmic framework for generating flow invariants from a collection of xMAS communication fabrics. For the sake of brevity, we call them *Chatterjee-Kishinevsky (CK) invariants*. They were shown to be crucial for timely convergence of safety verification experiments on the target xMAS communication fabrics. We revisit the problem that Chatterjee *et al.* addressed and provide an alternative recipe to generate similar invariants. The merit of our alternative lies in its rigorous algebraic analysis of the target systems which was not addressed in [Chatterjee and Kishinevsky, 2010a] or in [Chatterjee and Kishinevsky, 2012]. Our analysis provides a novel link between the discovered invariants and the network topology of the underlying communication fabrics and hence enhances our understanding about the CK invariants. We motivate the utility of the CK invariants with an example as follows:

CK Invariants and Property Verification: Consider the schematic diagram of a credit-based flow-control unit shown in Figure 6.1. It is implemented as a sequential circuit with 4 primary inputs and 18 flip-flops. To prove a simple safety property on it, we tried the bit-level hardware verification tool ABC [Mishchenko, 2013]. ABC's induction engine failed, but its interpolation engine proved it in 4.89 seconds. The proof was quick, but considering the small size of the design, it was slower than expected for ABC. However, when we provided an additional fact about the design that $num(B_1) + num(B_2) = num(B_3)$, where $num(B)$ is the number of flits in finite FIFO buffer $B \in \{B_1, B_2, B_3\}$ at any given point in time, ABC ran 100 times faster. It's induction engine proved the property in 0.04 seconds - an order of speedup that we need in practice for realistic problems. Such additional information about the system behavior, called *design invariants*, is crucial for the timely convergence of formal verification tools. [1] In this

[1] see [Chatterjee and Kishinevsky, 2010a] for another case-study revealing similar experience that without design invariants, verification took prohibitively long time even for trivial communication fabrics

chapter, we address the question of how to derive such design invariants automatically from communication fabrics.

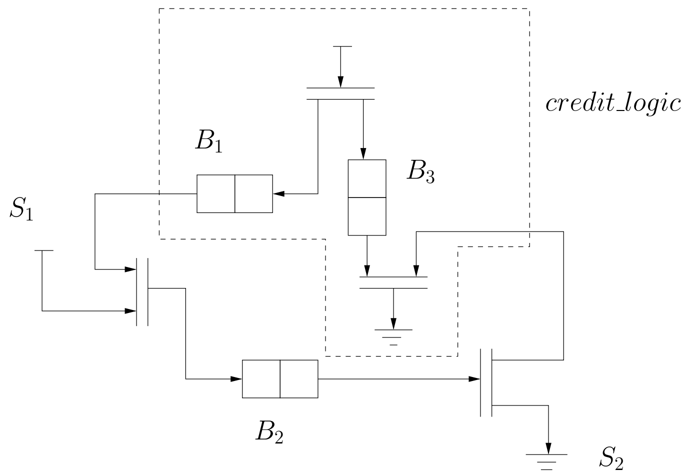


Figure 6.1: Credit mechanism

The invariant $num(B_1) + num(B_2) = num(B_3)$ mentioned above was discovered by a high-level analysis algorithm proposed by Chatterjee and Kishinevsky in [Chatterjee and Kishinevsky, 2010a]. It works at the micro-architecture level and discovers a set of non-trivial *linear relations* among the number of flits that the buffers in the fabric can hold at any given time. We refer to these relations as *CK invariants*. Since the seminal work on linear constraint generation by Cousot et al. [Cousot and Halbwachs, 1978], numerous other works have been published on generating linear invariants for various hardware and software systems. But to the best of our knowledge, [Chatterjee and Kishinevsky, 2010a] is the first work that focuses on a model of computation that is dedicated to hardware communication fabrics. However, the CK algorithm does not reveal any topological property of the fabrics responsible for such invariants and there was no insight given as to when their algorithm might yield interesting invariants. 1. Thus the CK algorithm is fortified with our mathematical analysis. Our analysis leads to an alternative algorithm for generating the same set of invariants. It is simple, yet more rigorous. We provide experimental results that demonstrate the impact of these invariants on both safety and liveness verification of communication fabrics.

A very effective new algorithm IC3 [Bradley, 2011], published after the CK algorithm, constructs precise invariants necessary to prove a particular safety property on a design. We show that the CK invariants complement those generated by IC3; the performance of IC3 can be enhanced by orders of magnitude, both for safety and liveness, if CK invariants are provided in advance. This underscores the usefulness of these invariants even further.

The chapter is organized as follows: Section 6.2 discusses related works. Section 6.3 introduces necessary mathematical notions used in the analysis. Section 6.4 presents our invariant generation algorithm and explains it with the help of an example. Experimental results are provided in Section 6.5 and Section 6.6 concludes the chapter.

6.2 Related Works

Automatic invariant generation is a holy grail of research in formal verification. This has been addressed from various perspectives and a huge volume of literature exists on this topic. Mining linear invariants is a particularly interesting sub-topic that has attracted attention, see for example [Cousot and Halbwachs, 1978], [Karr, 1976], [Gulwani and Necula, 2003]. Our work is most closely related to invariant generation for communication fabrics by Chatterjee et al. [Chatterjee and Kishinevsky, 2010a] and circuit theoretic analysis of marked graphs by Murata [Murata, 1977].

6.2.1 Work of Chatterjee et al.

[Chatterjee and Kishinevsky, 2010a] is the first work to identify the significance of linear invariants in formal verification of communication fabrics. However, it did not expose the connection between the generated invariants and the fabric topology. 1 1

6.2.2 Work of Tadao Murata

Murata [Murata, 1977] studied the role of null-spaces of matrices in reachability analysis of marked graphs. We expand on this in the following sense: our choice of the model of computation (xMAS) is more expressive than marked graphs, so direct application of Murata's analysis is not possible. We propose a transformation, called *type-specific sub-network derivation*, for the xMAS fabrics. It derives certain sub-networks from a given fabric. These sub-networks, after another transformation, conform to Murata's analysis and yield the same invariants as produced by the CK algorithm. 1. Other relevant works are those that generate linear or algebraic invariants for Petri nets [Sankaranarayanan *et al.*, 2003] [Clarisó *et al.*, 2005]. Since Petri nets are closely related to xMAS, it might be worth exploring potential synergy between our work and this body of Petri net research. This is left as a future work.

6.2.3 Relation with IC3-type invariant generation

A recent significant development in automatic invariant discovery is the algorithm IC3 by Aaron Bradley [Bradley, 2011]. This new paradigm has been followed by additional improvements [Een *et al.*, 2011] and extensions [Vizel *et al.*, 2012] [Hoder and Bjørner, 2012]. While these algorithms have impressive performance on our benchmarks, we demonstrate that their performances can be enhanced significantly if our structural invariants are provided as additional information. Bit-level analysis performed by IC3-type algorithms consume substantial time to discover topological invariants for our benchmarks which our high-level algorithm can discover in no time. 1

6.3 Preliminaries

6.3.1 Incidence Matrix and Fundamental Cycle matrix of a digraph

Below we review the concepts of the *incidence matrix* and the *fundamental cycle matrix* of a digraph. A simple example illustrating these concepts is also provided. The definitions in this section and the illustrative example at the end closely follow the exposition of [Deo, 1974].

Definition 5 (Cycle, Directed Cycle, Semicycle)**.** In a digraph G, a *directed cycle* is a *directed closed path*, *i.e.* an alternating sequence $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$ of vertices and edges such that e_i is a directed edge from v_i to v_{i+1} for $1 \leq i < n$, $v_1 = v_n$ and $v_i \neq v_j$ for any other pair of vertices on the path. A *semi-cycle* is a closed undirected path in G. It is an alternating sequence $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$ where e_i is an edge either from v_i to v_{i+1} or vice versa and no vertex is repeated except $v_1 = v_n$. A *cycle* is either a directed cycle or a semicycle.

Definition 6 (Incidence Matrix)**.** The *incidence matrix* of a digraph with n vertices, e edges and no self-loops is an $n \times e$ matrix $A = [a_{ij}]$, whose rows correspond to vertices and columns correspond to edges such that

$$
\begin{aligned}
a_{ij} &= 1, \text{ if } j\text{-th edge is incident out of } i\text{-th vertex} \\
&= -1, \text{ if } j\text{-th edge is incident into } i\text{-th vertex} \\
&= 0, \text{ if } j\text{-th edge is not incident on } i\text{-th vertex}
\end{aligned}
$$

Note that Definition 6 allows parallel edges in a digraph.

Example 2. Consider the directed graph shown in Figure 6.2(a). Incidence matrix A for this graph is the following:

$$
\begin{array}{c c}
 & \begin{array}{cccccccc} a & b & c & d & e & f & g & h \end{array} \\
\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} &
\begin{pmatrix}
0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
-1 & -1 & -1 & 0 & -0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & -1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

Definition 7 (Cycle Matrix)**.** Let G be a directed graph with e edges and q cycles (directed cycles or semicycles). An arbitrary orientation (clockwise or counterclockwise) is assigned to

(a) A typical directed graph

(b) A spanning tree (in bold) of the graph

Figure 6.2: A directed graph and one of its spanning tree

each of the q cycles. Then a *cycle matrix* $B = [b_{ij}]$ of the digraph G is a $q \times e$ matrix defined as

$$
\begin{aligned}
b_{ij} &= 1, && \text{if } i\text{-th cycle includes the } j\text{-th edge and the orientations} \\
&&& \text{of the edges and cycles coincide,} \\
&= -1, && \text{if } i\text{-th cycle includes the } j\text{-th edge,} \\
&&& \text{but the orientations of the two are opposite,} \\
&= 0, && \text{if the } i\text{-th cycle does not include the } j\text{-th edge}
\end{aligned}
$$

Example 3. The cycle matrix B for the graph in Figure 6.2(a):

$$
\begin{array}{c}
\\ q_1 \\ q_2 \\ q_3 \\ q_4
\end{array}
\begin{array}{c}
\begin{array}{cccccccc} a & b & c & d & e & f & g & h \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 1 & -1 & -1 & -1 & 0 & 0 \\
-1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

Here, the cycles are indexed by q_i's as follows:

$$
q_1 = v_1, f, v_2, g, v_5, d, v_1; \quad q_2 = v_2, g, v_5, c, v_4, e, v_2;
$$
$$
q_3 = v_1, f, v_2, e, v_4, c, v_5, d, v_1; \qquad q_4 = v_4, a, v_6, b, v_4;
$$

q_1 is a directed cycle, while q_2, q_3, and q_4 are semicycles.

Theorem 7. [Deo, 1974] Let B and A be the circuit matrix and incidence matrix of a self-loop-free digraph, such that the columns in B, and A are arranged using the same order of edges. Then

$$A \cdot B^T = B \cdot A^T = 0$$

where superscript T denotes the transpose matrix.

Definition 8 (Fundamental Cycle)**.** Consider a spanning tree T in a connected graph G. Adding any one chord to T will create exactly one cycle. Such a cycle, formed by adding a chord to a spanning tree, is called a *fundamental cycle*.

Example 4. The fundamental cycle matrix B_f for the same graph with respect to the spanning tree shown in Figure 6.2(b):

$$\begin{array}{c} \\ q_4 \\ q_3 \\ q_2 \end{array} \begin{array}{c} \begin{array}{cccccccc} b & d & g & a & c & e & f & h \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -1 & 0 & 0 \end{pmatrix} \end{array}$$

Since any spanning tree of G has $n-1$ edges, the number of fundamental cycles of G is $\mu = e-n+1$. Note that G could be a directed graph, but the spanning tree T is constructed disregarding the edge directions. Therefore, the fundamental cycles could be either directed cycles or semicycles.

Definition 9 (Fundamental Cycle Matrix)**.** A $\mu \times e$ cycle matrix B_f of a digraph G, for $\mu = e-n+1$, is called a *fundamental cycle matrix* if each of the μ rows corresponds to a fundamental cycle made by a chord (with respect to some specified spanning tree).

Corollary 1. $A \cdot B_f^T = B_f \cdot A^T = 0$

We use a few basic concepts from linear algebra, namely *linear independence*, *null space* of a matrix and *reduced row-echelon form* of a matrix and apply them to fundamental cycle matrices of digraphs. See e.g. [Strang, 2006] for more detail.

6.3.2 Marked Graphs, Reachability, Null Space Analysis

A *Marked graph* [Murata, 1977] is a well-studied formalism for modeling concurrent systems. Although xMAS fabrics are strictly more expressive than marked graphs, they may be abstracted as marked graphs under special circumstances. This allows the existing results on marked graphs to be applied to xMAS fabrics. We review some definitions and properties of marked graphs from [Murata, 1977].

Definition 10 (Simple Marked Graph)**.** A ***simple marked graph*** is a digraph with a non-negative number of *flits* assigned to each edge.

A vertex is ***firable*** if each of its incoming edges has at least one flit. The ***firing*** of a firable vertex consists of removing one flit from each of its incoming edges, and adding one flit to each of its outgoing edges. A vertex with no incoming edge is a *source* and a vertex with no outgoing edges is a *sink*.

Suppose G is a marked graph with p edges and t vertices. A *marking* or a *state vector* M_k of G is a $p \times 1$ column vector of non-negative integers, the j-th entry denoting the number of flits on edge j immediately after the k-th firing. M_0 denotes the initial marking (or the initial condition). The *elementary firing vector* U_k is a $t \times 1$ Boolean column vector containing a 1 in the positions corresponding to the vertices fired at the k-th firing.

Consider a sequence of firing vectors $\bar{U} = U_1, U_2, \ldots$ and suppose a fabric starts from an initial state M_0 and the sequence of states $M_1, M_2, \ldots$ is such that the transition from state M_{t-1} to state M_t is the result of activity U_t. For $t \geq 1$, we may write

$$M_t = M_{t-1} + A^T U_t, \quad t = 1, 2, \ldots \tag{6.1}$$

where A^T is the transpose of the incidence matrix. For the first n activities $\bar{U}[1, n] = U_1, U_2, \ldots, U_n$, we may add n instances of (6.1) to get:

$$M_n = M_0 + A^T . \left(\sum_{t=1}^{n} U_t \right) \tag{6.2}$$

We use the abbreviation $\Delta_n(M) = A^T \Sigma_n$ to denote the above equation where $\Delta_n(M) = M_n - M_0$ and $\Sigma_n = \sum_{t=1}^{n} U_t$. Using Corollary 1, we obtain:

$$B_f . \Delta_n(M) = B_f .(A^T . \Sigma_n) = (B_f . A^T) . \Sigma_n = 0.$$

In other words, vector $\Delta_n(M)$ belongs to the null-space of B_f. Therefore, if M_n is reachable from M_0 for any possible activity sequence $\bar{U}$, the vector $M_n - M_0$ necessarily belongs to the null-space of B_f. Since $M_0 = \mathbf{0}$ in our application, then M_n belongs to the null-space of B_f for any $n > 0$. Hence the following theorem holds:

Theorem 8. The null-space of B_f is an over-approximation of the reachable state space.

Any marked graph of interest would have one or more non-zero reachable states. Thus the null space of B_f subsumes $\{\mathbf{0}\}$ properly and hence the column vectors of B_f *cannot be linearly independent*. The linear dependence of the column vectors of B_f can be obtained by analyzing the reduced row-echelon form of B_f.

Theorem 9. The dependence relationships obtained by the reduced row-echelon analysis of B_f yields a set of reachability invariants for marked graph G.

6.4 Invariant Mining For xMAS fabrics

As a model of computation, xMAS is different from a simple marked graph. It can represent many design idioms effectively and succinctly. Unfortunately, the simple relation of Equation (6.1) is not sufficient to capture the transition relation of an arbitrary xMAS fabric. As a result, we cannot apply Theorem 9 directly to xMAS. While there are compelling structural similarities between xMAS and simple marked graphs, their fundamental semantic differences are the following:

- The firing of a vertex of a marked graph only requires the presence of flit on all its incoming edges; a marked graph has no notion of conditional firing based on the type of flits on the edges. In contrast, xMAS has components (switches, arbiters) whose firing depends on the type of flits at their incoming channels.
- In a marked graph, the firing of a vertex does not depend on the presence or absence of flits on its outgoing edges, but in xMAS, a component cannot fire if a buffer on its output channel(s) is not ready to accept a new flit.

However, for generating structural invariants, we might assume that the buffers are of unbounded capacity. Under this assumption, the following important family of xMAS fabrics can be expressed as simple marked graphs:

Proposition 1. The behavior of an xMAS fabric with *unbounded* buffers, that has no switch, or arbiter, can be modeled as a marked graph.

1 We illustrate the notions of type-specific sub-networks, derivation of simple marked graphs, and other intermediate steps to generate invariants in the following sections using the example of $\mathcal{VCO}$.

6.4.1 Type-specific sub-network derivation

Different types of flits can travel across xMAS fabrics. These fabrics are so designed that a particular sub-network (*i.e.* a subset of buffers, channels and some other xMAS components) are responsible for steering flits of one particular type from their sources to their destinations. These sub-networks, each steering one type of flit, need not be structurally disjoint. They can share buffers, channels, or other structural components with sub-networks handling other types of flits. A flit can traverse different sub-networks designated for different types of flits because its own type may change enroute to its destination. Below we illustrate this notion of type-specific sub-network with the help of $\mathcal{VCO}$.

Example 5. Consider the fabric $\mathcal{VCO}$ shown in Figure 6.3. It models a virtual channel that enforces ordering among flits. Here we identify the type-specific sub-networks of $\mathcal{VCO}$ as follows: $\mathcal{VCO}$ carries two types of flits, *viz.* $type(A_1)$ and $type(A_2)$ coming from sources A_1 and A_2 respectively. Buffers B_3, B_5 and B_6 host $type(A_1)$-flits as they move to their destination, while B_1 and B_2 act as auxiliary buffers controlling their flow. These five buffers, along with other components, form the sub-network for $type(A_1)$-flits in $\mathcal{VCO}$. Figure 6.4 depicts this sub-network. Similarly, Figure 6.5 depicts the sub-network that steers $type(A_2)$-flits. We use the *un-ordered triplet* notation (x, y, z) to denote various three-terminal xMAS components of $\mathcal{VCO}$, where x, y, z are names of the channels associated with the component. We see that arbiter (d_1, d_2, e), fork (e, k_1, k_2), switches (k_1, k_3, k_4) and (r_1, r_2, r_3), buffer B_5 and channels e, k_1, k_2, r_1 are shared between these two sub-networks. Note in Figures 6.4 and 6.5 how switches and arbiters are structurally broken according to their roles in steering flits

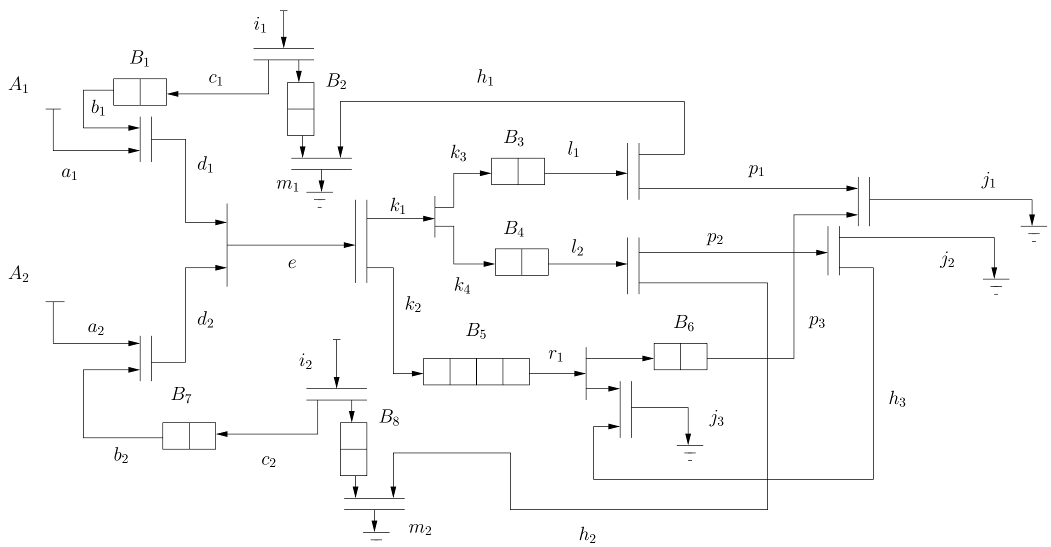


Figure 6.3: $\mathcal{VCO}$: A virtual channel with ordering

associated with the corresponding sub-networks. A simple graph crawling algorithm, starting from the sources and traveling towards the sinks, may discover these sub-networks easily. We leave the problem of designing details of such a graph-crawling algorithm for the future.

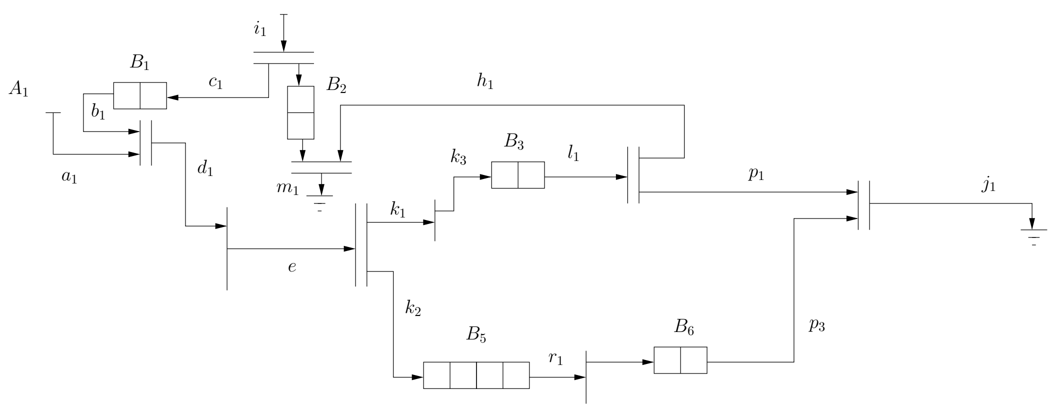


Figure 6.4: Sub-network that steers $type(A_1)$-flit

Removal of broken components: During the construction of these sub-networks, we only break switches, and arbiters by selectively disconnecting some of their outgoing or incoming channels. With such channel disconnections, these xMAS components cease to work as conditional decision-makers; rather they behave simply as a direct connection between their (remaining) input channels and (remaining) output channels. We can, therefore, short these broken switches and arbiters in the resulting sub-networks, and eliminate them. Such

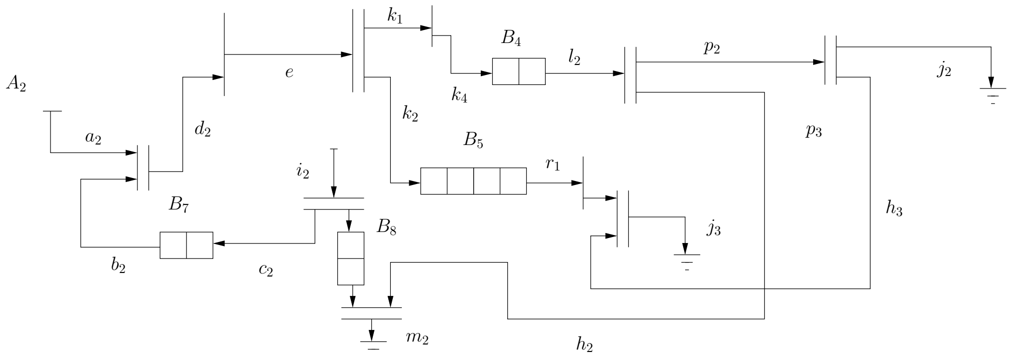


Figure 6.5: Sub-network that steers $type(A_2)$-flit

modified sub-networks, with the broken components shorted, are shown in Figure 6.6 and Figure 6.7 respectively. Therefore, a type-specific sub-network does not contain switches and arbiters.

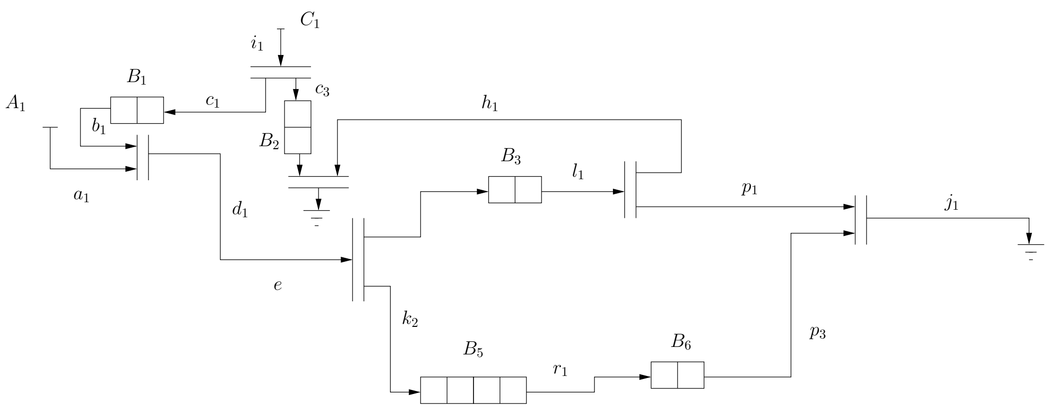


Figure 6.6: Sub-network of Figure 6.4 simplified by 'shorting' arbiters and switches

6.4.2 Sub-networks as marked graphs, Derivation of Invariants

The ability to make conditional decisions using switches and arbiters creates xMAS networks more expressive than simple marked graphs. However, the type-specific sub-networks do not contain switches and arbiters so their behavior can be modeled as marked graphs. Figures 6.8(a) and 6.8(b) denote the marked graphs for the sub-networks of Figures 6.6 and Figure 6.7 respectively. These two graphs are similar in structure, but their vertices and transitions represent different components and activities of $\mathcal{VCO}$ as discussed in Table 6.1 and 6.2.

Since both the marked graphs have the same cyclic core induced by the vertices $\{v_1, v_2, v_3\}$ and $\{u_1, u_2, u_3\}$ respectively, we consider (without any loss of generality) that they have the

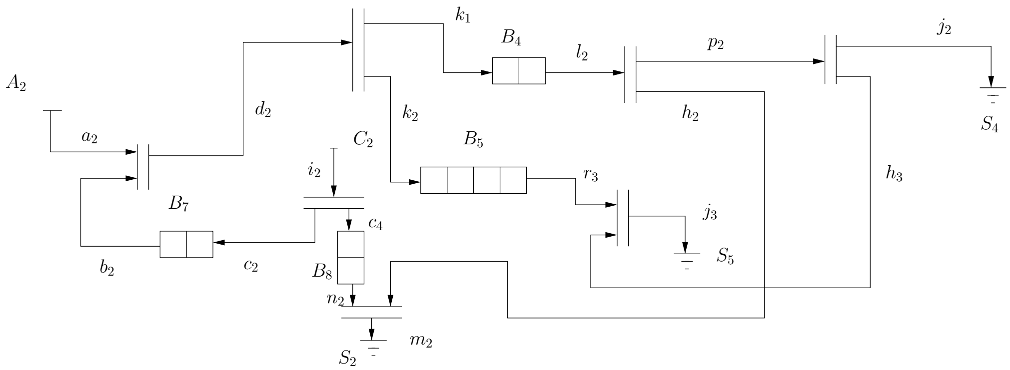


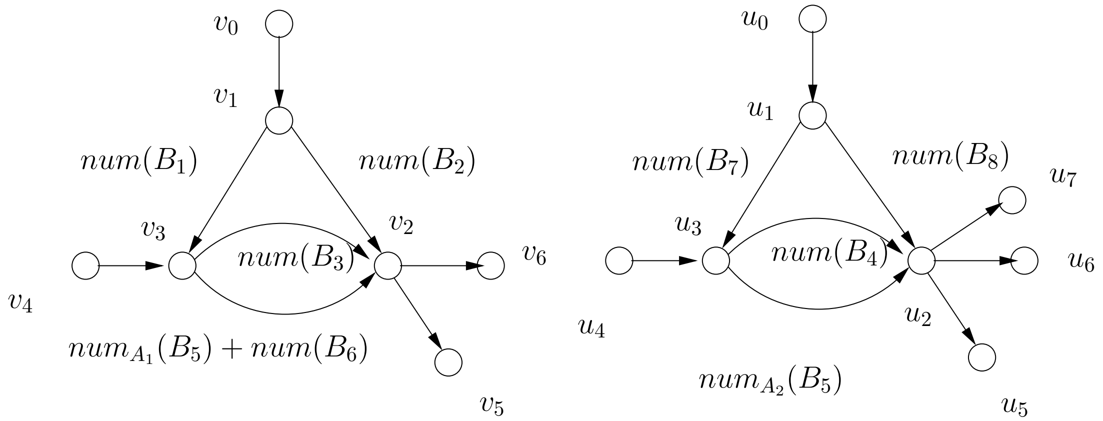
Figure 6.7: Sub-network of Figure 6.5 simplified by 'shorting' arbiters and switches

Marked Graph Vertex	Corresponding logic in sub-network
v_0	source C_1
v_1	fork (i_1, c_1, c_3)
v_2	fork (l_1, p_1, h_1), join (h_1, m_1, n_1), join (p_1, p_3, j_1)
v_3	join (a_1, b_1, d_1), fork (e, k_1, k_2)
v_4	source A_1
v_5	sink S_1
v_6	sink S_3

Table 6.1: Correspondence between vertices of marked graph in Figure 6.8(a) and xMAS components of Figure 6.6

Marked Graph Vertex	Corresponding logic in sub-network
u_0	source C_2
u_1	fork (i_2, c_2, c_4)
u_2	fork (l_2, p_2, h_2), fork (p_2, h_3, j_2), join (h_2, m_2, n_2), join (h_3, r_3, j_3)
u_3	join (a_2, b_2, d_2), fork (e, k_1, k_2)
u_4	source A_2
u_5	sink S_2
u_6	sink S_5
u_7	sink S_4

Table 6.2: Correspondence between vertices of marked graph in Figure 6.8(b) and xMAS components of Figure 6.7



(a) Marked Graph for Figure 6.6 (b) Marked Graph for Figure 6.7

Figure 6.8: Marked graphs for the sub-networks of $\mathcal{VCO}$

same fundamental cycle matrix B_f, as well as the same reduced row-echelon form $rref(B_f)$ as follows:

$$B_f = \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 1 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \end{array}, \quad rref(B_f) = \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} \end{array}$$

Columns of B_f and $rref(B_f)$ are labeled with a, b, c and d and their correspondence to the edges of the marked graphs of Figures 6.8(a) and 6.8(b) are shown in Table 6.3. The other vertices and edges of the marked graphs have no contribution to B_f and $rref(B_f)$; hence are omitted from the matrices. $rref(B_f)$ is derived from B_f using a standard algorithm implemented in an off-the-shelf linear algebra package.

Edge Label	Edge in Fig. 6.8(a)	Edge in Fig. 6.8(b)
a	(v_1, v_3)	(u_1, u_3)
b	$(v_3, v_2)_{upper}$	$(u_3, u_2)_{upper}$
c	(v_2, v_1)	(u_2, u_1)
d	$(v_3, v_2)_{lower}$	$(u_3, u_2)_{lower}$

Table 6.3: Edge identifiers for the marked graphs

As mentioned in the last section, the null space of B_f (and hence of $rref(B_f)$) is not just a singleton $\{\mathbf{0}\}$. This makes the column vectors of B_f as well as of $rref(B_f)$ linearly dependent. This underlying linear dependence relation among the columns is evident from $rref(B_f)$ and it may be expressed using the following relations

$$\begin{aligned} a - c + d &= 0 \\ b - d &= 0 \end{aligned}$$

where a and b are treated as pivot variables; c and d as free variables. We may replace the column labels a, b, c and d in the above relations with the edge markings from Figure 6.8(a) and 6.8(b). After suitable rearrangement, we get the following relations:

$$\text{Fig. 6.8(a)} \begin{cases} num(B_2) = num(B_1) + num_{A_1}(B_5) + num(B_6) \\ num(B_3) = num_{A_1}(B_5) + num(B_6) \end{cases}$$

$$\text{Fig. 6.8(b)} \begin{cases} num(B_8) = num(B_7) + num_{A_2}(B_5) \\ num(B_4) = num_{A_2}(B_5) \end{cases}$$

The above relations serve as invariants to the reachable state space of $\mathcal{VCO}$. We discuss their roles in model checking of properties of $\mathcal{VCO}$ in detail in the next section.

We applied this technique of type-specific sub-network derivation on the other benchmarks like $\mathcal{VC}$, $\mathcal{VCB}$ etc. and generated invariants from them. This way, we managed to generate all invariants that were generated by the original CK algorithm as reported in [Chatterjee and Kishinevsky, 2010a]. The generated invariants are tabulated in Table 6.4.

6.5 Experimental Results

We used ABC [Mishchenko, 2013] as our safety verification environment. Experiments were performed on a laptop with 1.2 GHz Intel Celeron processor and 2 GB RAM. We present in Table 6.5 run-times (in seconds) taken by ABC on various models considered.

The results in Table 6.5 pertain to three experiments, each involving three different verification engines (hence, a total of nine columns of run-times). The first three columns show the name of the design, number of primary inputs (#pi) and number of flip-flops (#ff) respectively. The results of the first experiment span columns 4, 5 and 6. Here we used three different engines, *induction* [van Eijk, 2000], *interpolation* [McMillan, 2003] and *property directed reachability* (PDR in short) [Bradley, 2011] to demonstrate the effectiveness of the invariants of Table 6.4 on the respective designs. We used ABC's implementations of the algorithms which are available as commands `dprove`, `int`, and `pdr` respectively and used ABC's default resource limits. Note that command `dprove` applies sequential simplification on the underlying circuit before calling the verification engines. This is often a key step to make the proof converge. The runs which could not decide validity of a property within the default resource limits of ABC are marked with '-'. It turns out that the interpolation engine (as applied to un-simplified designs through the command `int`) times out most of the time. We believe that ineffectiveness of interpolation, which is a powerful proof engine otherwise, on our benchmarks demonstrates the inherent difficulty of verification for these apparently small designs. A similar observation was reported in [Chatterjee and Kishinevsky, 2010a]

model	BUFFER RELATIONS
buffered vc	$num(B_1) + num(B_3) + num_{A_1}(B_{ch}) = num(B_2)$ $num(B_5) + num(B_4) + num_{A_2}(B_{ch}) = num(B_6)$
ordered vc	$num(B_1) + num(B_3) = num(B_2)$ $num(B_7) + num(B_4) = num(B_8)$ $num(B_3) = num_{A_1}(B_5) + num(B_6)$ $num(B_4) = num_{A_2}(B_5)$ $num(B_5) = num_{A_1}(B_5) + num_{A_2}(B_5)$
scoreboard	$num(B_1) + \sum_{i=5}^{9} num(B_i) = num(B_3)$ $num(B_2) + \sum_{i=10}^{14} num(B_i) = num(B_4)$

Table 6.4: Assumptions used in the proofs

too. PDR has been demonstrated over a wide variety of designs to be the strongest proof engine and here it proves all the properties quite fast. But the remarkable observation is that in many cases, the properties can be proved using one-step induction (under `dprove`), sometimes beating PDR by a large margin.

Column 7, 8 and 9 show the run-times of `dprove`, `int` and `pdr` when we try to prove a candidate safety property φ_{nb} using the invariants of Table 6.4 as constraints. Property φ_{nb} specifies that channel e (see Figure 6.3) is non-blocking. In LTL, $\varphi_{nb} := \mathbf{G}(e.irdy \Rightarrow e.trdy)$. We simply asked the proof engines to prove all the invariants and φ_{nb} together. Columns 10, 11 and 12 report run-times when the same engines are used to prove φ_{nb} alone, without the invariants. In this round of experiments, induction and interpolation failed most of the time, while PDR managed to derive all the proofs. For each design, we mark a cell with '*' to indicate which engine provided the best run-time. The overall table and the cells with '*' show a clear advantage when invariants are used. In some cases, run-times are rather counter-intuitive. For example, PDR took longer for VCB and Master/Slave to prove φ_{nb} when the invariants were enabled. Interestingly, induction (column 7) did remarkably well for those cases. This underscores the benefit of our high-level null space analysis. PDR possibly infers equivalent invariants with its bit-precise state-space analysis, but at a cost of extra run-time.

It may be noted that in the cases of time-outs, we used ABC's default resource limits only. We provide the actual times for which those engines ran in Table 6.6. These show that the corresponding resource limits were reached by ABC quite early; one could possibly make those runs successful by increasing ABC resources. Since some other engine succeeded even under default resource limits on those cases, we did not pursue any further tuning of ABC's resource limits in those experiments.

design	#pi	#ff	Proving Invariants Only			Proving Target Property with Invariants			Proving Target Property without Invariant		
			induction (sec.)	interpolation (sec.)	pdr (sec.)	induction (sec.)	interpolation (sec.)	pdr (sec.)	induction (sec.)	interpolation (sec.)	pdr (sec.)
VC	11	82	0.03	3.72	0.10	**0.04***	9.15	0.10	0.89 (int)	4.01	0.23
VCB	12	93	0.06	-	106.45	**0.05***	-	50.91	-	-	33.46
VCO	13	101	0.11	-	0.34	**0.10***	-	0.35	-	-	8.28
VCO-VCO	19	192	-	-	1.65	-	-	**2.07***	-	-	392.94
Master/Slave	16	152	0.16	-	7.23	**0.18***	-	425.74	-	-	48.39
scoreboard	23	133	-	-	5.70	-	-	**7.28***	-	-	30.19

Table 6.5: Experiment on various communication fabrics (1)

design	#pi	#ff	Proving Invariants Only			Proving Target Property with Invariants			Proving Target Property without Invariant		
			induction (sec.)	interpolation (sec.)	pdr (sec.)	induction (sec.)	interpolation (sec.)	pdr (sec.)	induction (sec.)	interpolation (sec.)	pdr (sec.)
VC	11	82	0.03	3.72	0.10	**0.04***	9.15	0.10	0.89 (int)	4.01	0.23
VCB	12	93	0.06	TO(26.48)	106.45	**0.05***	TO(15.28)	50.91	TO(12.90)	TO(26.88)	33.46
VCO	13	101	0.11	TO(30.40)	0.34	**0.10***	TO(36.21)	0.35	TO(26.60)	TO(20.68)	8.28
VCO-VCO	19	192	TO(8.27)	TO(10.93)	1.65	TO(9.83)	TO(10.56)	**2.07***	TO(6.80)	TO(32.09)	392.94
Master/Slave	16	152	0.16	TO(21.32)	7.23	**0.18***	TO(16.54)	425.74	TO(6.16)	TO(8.75)	48.39
scoreboard	23	133	TO(18.78)	TO(13.48)	5.70	TO(44.79)	TO(12.3)	**7.28***	TO(13.54)	TO(13.16)	30.19

Table 6.6: Experiment on various communication fabrics (2)

6.6 Conclusion

1. The latter essentially applies **Kirchhoff's voltage law** (KVL) on the loops of communication fabrics. This demonstrates the influence of feedback connections on the invariants which was not revealed by the original version of the CK algorithm. Our observation that the CK invariants are feedback-induced invariants highlights the fact that they are only one particular kind of invariant that strengthens inductive proofs. We have observed that there are other invariants that can improve verification speed even further. For example, one such helpful invariant for $\mathcal{VCB}$ of Figure 3.7 is $0 \leq num(B_{ch}) \leq 1$ for the given buffer sizes. However, our KVL based analysis cannot find such invariants because they are not directly dependent on the feedback structure of the fabric. Since implementation or pseudocode of the original CK algorithm is not publicly available, it is hard to predict whether it can find such invariants. An interesting research topic, therefore, would be to find techniques that can discover such invariants beyond the feedback-induced ones.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We have developed a bit-level liveness verification framework for sequential hardware systems. Our framework has a proof engine and a bug finder. While in theory our framework is capable of solving general liveness obligations, we found that it is mostly an intractable problem in practice. We targeted a particular liveness property called response property, and a family of industrially relevant communication fabric designs. We demonstrated that monolithic liveness verification algorithms can easily fail to converge on such designs. To mitigate this scalability problem, we proposed various heuristics. As part of this scheme, we discovered a general pattern in the behavior of communication fabrics. We called this pattern as ranking structure and demonstrated its influence in scalable liveness verification. In the course of this research, we have identified several interesting questions that need be answered to bring further scalability and automation in liveness verification. Some of them are outlined below as topics for future investigation.

7.2 Future Work

- In our case studies, we have focussed mainly on *credit-based flow-control systems*. The logical pattern associated with the ranking structures of such systems indicates that such ranking structures may be inferred from other communication fabrics as well as from other general hardware designs. However, we have kept the scope of our experiments limited to our benchmarks. Further generalization of the discovered ranking structures is left as future work.

- We have chosen the k-LIVENESS algorithm as our vehicle of experimentation with the idea of disjunctive stabilizing assertion. We believe that stabilizing assertions can accelerate other off-the-shelf liveness verification algorithms, for example algorithms based on BDD construction or liveness-to-safety transformation. Whether the ranking structures (of communication fabrics) can accelerate these other algorithms and which algorithm would be the most effective one are interesting research questions. We leave these studies as future work.

- We observed that impressive run-time has resulted from using the most relevant stabilizing assertions obtained from the ranking structures derived manually from the fabrics. It might be possible to construct a high-level algorithm that can leverage the if-then-else reasoning structure associated with a fabric and produce a concise ranking structure automatically. We believe that it is an interesting and important open problem and leave this for future exploration.

- Fabrics are often constructed by composing two or more smaller fabrics. We observed that in some cases the ranking structures of the smaller fabrics can be composed to generate the ranking structure of the larger fabric. We believe that some interesting compositional reasoning algorithm can be designed in this space and the topic should be investigated further. As an example, we note that Holcomb *et al.* proposed a SAT-based compositional technique for solving bounded liveness problem for communication fabrics [Holcomb *et al.*, 2012]. It would be interesting to generalize such approaches and unify them with the idea of ranking structure.

- We believe that our work on response verification has subtle conceptual connections to the deadlock verification approaches of [Verbeek and Schmaltz, 2011] and [Gotmanov *et al.*, 2011], but we leave any further exploration of this connection as future work.

Bibliography

[Abrahamson, 1980] K. Abrahamson. Decidability and expressiveness of logics of processes. In *PhD Thesis, University of Washington*, 1980.

[Arvind and Culler, 1986] Arvind and David E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[Baier and Katoen, 2008] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[Baumgartner and Mony, 2009] Jason Baumgartner and Hari Mony. Scalable liveness checking via property-preserving transformations. In *DATE*, pages 1680–1685, 2009.

[Ben-Amram, 2009] Amir M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 109–123, Berlin, Heidelberg, 2009. Springer-Verlag.

[Benveniste *et al.*, 2003] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[Biere *et al.*, 2002] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *In FMICSŠ02: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*. Elsevier, 2002.

[Bilsen *et al.*, 1995] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *IEEE Intl. Conf. Acoustics, Speech, and Signal Processing*, volume 5, pages 3255–3258, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[Bloem *et al.*, 2006] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *Form. Methods Syst. Des.*, 28(1):37–56, January 2006.

[Bradley *et al.*, 2011] Aaron Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *FMCAD*, Austin, TX, USA, 2011.

[Bradley, 2011] A. Bradley. Sat-based model checking without unrolling. In *VMCAI*. Springer Verlag, 2011.

[Bryant, 1992] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[Buck, 1993] J. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993.

[Chatterjee and Kishinevsky, 2010a] Satrajit Chatterjee and Michael Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In *CAV*. Springer Verlag, 2010.

[Chatterjee and Kishinevsky, 2010b] Satrajit Chatterjee and Michael Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In *CAV*, pages 321–338, 2010.

[Chatterjee and Kishinevsky, 2012] Satrajit Chatterjee and Michael Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. *Form. Methods Syst. Des.*, 40(2):147–169, April 2012.

[Chatterjee *et al.*, 2010] Satrajit Chatterjee, Mike Kishinevsky, and Umit Ogras. Quick formal modeling of communication fabrics to enable verification. In *HLDVT*, pages 42–49, 2010.

[Claessen and Sörensson, 2012] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *FMCAD*, pages 52–59, 2012.

[Clarisó *et al.*, 2005] Robert Clarisó, Enric Rodríguez-Carbonell, and Jordi Cortadella. Derivation of non-structural invariants of petri nets using abstract interpretation. In *Proceedings of the 26th international conference on Applications and Theory of Petri Nets*, ICATPN'05, pages 188–207, Berlin, Heidelberg, 2005. Springer-Verlag.

[Clarke *et al.*, 1999] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[Cook *et al.*, 2006] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. *SIGPLAN Not.*, 41:415–426, June 2006.

[Cook *et al.*, 2007] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. *SIGPLAN Not.*, 42:265–276, January 2007.

[Cook *et al.*, 2011] B. Cook, J. Fisher, E. Krepska, and N. Piterman. Proving stabilization of biological systems. In *VMCAI*. Springer Verlag, 2011.

[Cormen *et al.*, 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Cousot and Halbwachs, 1978] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[Dally and Towles, 2003] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[Deo, 1974] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science (Prentice Hall Series in Automatic Computation)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1974.

[Duato *et al.*, 2002] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[Een *et al.*, 2011] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, TX, 2011. FMCAD Inc.

[Emerson and Lei, 1986] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, pages 267–278, 1986.

[Emerson and Lei, 1987] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Sci. Comput. Program.*, 8:275–306, June 1987.

[Emerson, 1990] E. Allen Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[Francez and Kozen, 1984] Nissim Francez and Dexter Kozen. Generalized fair termination. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 46–53, New York, NY, USA, 1984. ACM.

[Gotmanov *et al.*, 2011] A. Gotmanov, S. Chatterjee, and M. Kishinevsky. Verifying deadlock-freedom of communication fabrics. In *VMCAI*. Springer Verlag, 2011.

[Gulwani and Necula, 2003] Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 74–84, New York, NY, USA, 2003. ACM.

[Hardin *et al.*, 2001] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using bdds. *Form. Methods Syst. Des.*, 18(2):131–140, March 2001.

[Hoder and Bjørner, 2012] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 157–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[Hojati *et al.*, 1993a] Ramin Hojati, Robert K. Brayton, and Robert P. Kurshan. Bdd-based debugging of design using language containment and fair ctl. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 41–58, London, UK, 1993. Springer-Verlag.

[Hojati *et al.*, 1993b] Ramin Hojati, Hervé J. Touati, Robert P. Kurshan, and Robert K. Brayton. Efficient omega-regular language containment. In *Proceedings of the Fourth International Workshop on Computer Aided Verification*, CAV '92, pages 396–409, London, UK, UK, 1993. Springer-Verlag.

[Holcomb *et al.*, 2012] Daniel Holcomb, Alexander Gotmanov, Michael Kishinevsky, and Sanjit A. Seshia. Compositional performance verification of noc designs. In *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2012.

[HWM, 2012] Hardware model checking competition, 2012: http://fmv.jku.at/hwmcc11/. 2012.

[Karr, 1976] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[Kesten *et al.*, 1998] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, ICALP '98, pages 1–16, London, UK, UK, 1998. Springer-Verlag.

[Kupferman and Vardi, 2005] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Proc. 46th IEEE Symp. on Foundations of Computer Science*, pages 531–540, Pittsburgh, October 2005.

[Lamport, 1980] Leslie Lamport. "sometime" is sometimes "not never": on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 174–185, New York, NY, USA, 1980. ACM.

[Lee and Messerschmitt, 1987] Edward A. Lee and David G. Messerschmitt. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *COMPCON*, pages 310–315, 1987.

[Lehmann *et al.*, 1981] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 264–277, London, UK, 1981. Springer-Verlag.

[Long *et al.*, 2011] J. Long, S. Ray, B. Sterin, A. Mishchenko, and R Brayton. Enhancing abc for stabilization verification of systemverilog/vhdl models. In *DIFTS*, 2011.

[Manna and Pnueli, 1991] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83:97–130, June 1991.

[Manna and Pnueli, 2010] Zohar Manna and Amir Pnueli. Time for verification. chapter Temporal verification of reactive systems: response, pages 279–361. Springer-Verlag, Berlin, Heidelberg, 2010.

[McMillan, 2003] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.

[Mishchenko, 2013] Alan Mishchenko. Abc verification system: http://www.eecs.berkeley.edu/~alanmi/abc/. 2013.

[Murata, 1977] Tadao Murata. Circuit theoretic analysis and synthesis of marked graphs. In *IEEE Transactions on Circuits and Systems*, volume 24, pages 400–405, 1977.

[Murata, 1989] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989.

[Pnueli, 1983] Amir Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 278–290, New York, NY, USA, 1983. ACM.

[Podelski and Rybalchenko, 2004] Andreas Podelski and Andrey Rybalchenko. Transition invariants. *Logic in Computer Science, Symposium on*, 0:32–41, 2004.

[Queille and Sifakis, 1983] J. P. Queille and J. Sifakis. Fairness and related properties in transition systems Ů a temporal logic to deal with fairness. In *Acta Informat*, pages 195–220, 1983.

[Ravi *et al.*, 2000] Kavita Ravi, Roderick Bloem, and Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 143–160, London, UK, UK, 2000. Springer-Verlag.

[Sankaranarayanan *et al.*, 2003] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Petri net analysis using invariant generation. In *Verification: Theory and Practice, LNCS 2772:682Ű701*, pages 682–701. Springer Verlag, 2003.

[Schuppan and Biere, 2004] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *Int. J. Softw. Tools Technol. Transf.*, 5(2):185–204, 2004.

[Somenzi *et al.*, 2002] Fabio Somenzi, Kavita Ravi, and Roderick Bloem. Analysis of symbolic scc hull algorithms. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, FMCAD '02, pages 88–105, London, UK, UK, 2002. Springer-Verlag.

[Strang, 2006] Gilbert Strang. *Linear ALgebra and Its Applications, Fourth Edition*. Thomson Brooks/Cole, Inc., 2006.

[Tarjan, 1972] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Thomas, 1990] Wolfgang Thomas. Handbook of theoretical computer science (vol. b). chapter Automata on infinite objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.

[van Eijk, 2000] C. A. J. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(7):814–819, 2000.

[Vardi and Wolper, 1984] Moshe Y. Vardi and Pierre Wolper. Automata theoretic techniques for modal logics of programs: (extended abstract). In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, STOC '84, pages 446–456, New York, NY, USA, 1984. ACM.

[Verbeek and Schmaltz, 2011] F. Verbeek and J. Schmaltz. Huntng deadlock efficiently in micro-architectural models of communication fabrics. In *FMCAD*, 2011.

[Vizel *et al.*, 2012] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy abstraction and sat-based reachability for hardware model checking. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD'12, Austin, TX, 2012. FMCAD Inc.

[Wolper *et al.*, 1983] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, SFCS '83, pages 185–194, Washington, DC, USA, 1983. IEEE Computer Society.

[Xie and Beerel, 1999] Aiguo Xie and Peter A. Beerel. Implicit enumeration of strongly connected components. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, ICCAD '99, pages 37–40, Piscataway, NJ, USA, 1999. IEEE Press.

[Zhang and Malik, 2002] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *CAV*, pages 17–36, 2002.